Measuring Object-Orient Software Products

Clark Archer
Winthrop University

June 1995

1. REPORT DATE **JUN 1995**	2. REPORT TYPE	3. DATES COVERED **00-00-1995 to 00-00-1995**

4. TITLE AND SUBTITLE **Measuring Object-Orient Software Products**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University,Software Engineering Institute,Pittsburgh,PA,15213**	8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT **Same as Report (SAR)**	18. NUMBER OF PAGES **83**	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**			

This document was prepared for the

SEI Joint Program Office
HQ ESC/ENS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD
position. It is published in the interest of scientific and technical information
exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

Preface

This module provides an overview of the merging of a paradigm and a process, the object-oriented paradigm and the software measurement process. The concept of a measure and the process of measurement are discussed briefly, followed by a presentation of the issues raised by object-oriented software development.

Capsule Description

Scope

The concept that software systems and the associated software development process constitute an engineering discipline is gaining acceptance. It is also clear that measurement is necessary for this software development process to be successful. The recent movement toward object-oriented technology has added another level of complexity to the software engineering discipline. Attempts to measure both software products and the software development process have produced what are currently called 'metrics.' Many such 'metrics' have been proposed; most of these have been defined and then tested in an artificial or restricted environment. No set of standards for accessing these 'metrics' has been developed and been universally accepted. As can be seen in the annotated outline (starting on page two), the term 'measure' is preferred to the term metric for the software product measures that have been proposed so far.

When measuring object-oriented software products, there are two key issues that need to be addressed: (1) measuring and the resultant measure, and (2) the object-oriented paradigm. Neither of these issues has been satisfactorily resolved in the software engineering and computer science literature.

Exactly what constitutes a measure is still an issue of contention; and complexity, which on the surface appears to be a simple property, has spawned a variety of interpretations. As a result, the same measure has been used to describe different interpretations of the same property in a software product. This module addresses the need to establish properties of a measure and discusses attempts to set a minimal set of requirements for a measure.

Many approaches to developing object-oriented software have been presented in the literature and each approach has introduced different terminology. A list of terms for the object-oriented paradigm is introduced in the annotated outline to provide a common arena for presenting the object-oriented paradigm.

Philosophy

As stated in the section above, measuring object-oriented software products has a multitude of problems. This module addresses these problems by:

- indicating the state of the practice of object-oriented measures

- suggesting a set of terminology for the object-oriented paradigm

- suggesting a minimal set of measurable features of object-oriented software products

- indicating the present diversity of measures that have been proposed for object-oriented software products

- giving examples of the design and coding of several problems and give a suite of measures for each example.

Establishing a common vocabulary for the object-oriented paradigm and a minimal set of standards for a software measure will aid the development of future measures and the refinement of standards. It is to this end that this module is written.

Acknowledgments

The author would like to acknowledge Jorge Díaz-Herrera and Gary Ford at the SEI for their technical assistance and advice; Nancy Mead, Linda Northrop, and Carol Sledge at the SEI for their valuable reviews of this paper; Jack Hilbing for his support; and Rachel Haas for her invaluable editorial assistance.

Author's Address

Comments on this module are solicited, and may be sent to the SEI Software Engineering Institute Community Sector or to the author:

Clark B. Archer
Department of Computer Science
Winthrop University
Rock Hill, SC 29733
Internet: archerc@winthrop.edu

Table of Contents

List of Tables

List of Figures

Measuring Object-Oriented Software Products

Outline

1. Measurement - An Overview

It is quite clear that measurement is necessary for the software development process to be successful. In addition, the path to controlling and improving the software design process may lie in the use of an object-oriented design approach. The recent movement toward object-oriented technology must also include the processes that control object-oriented development, namely software measures. Tom DeMarco summarizes the essence of these sentiments by stating, "You cannot control what you cannot measure" [DeMarco 87]. Measurement encompasses many aspects of the software life cycle. The emphasis of this document is on the design and implementation phases of an object-oriented approach.

Viewing measurement from a higher level, software measurement activities must have specific objectives. After these objectives are identified, the concepts, terminology, and measures presented in this module can be used to construct a framework applicable to the environment under consideration. One such objective-oriented approach is the Goal-Question-Metric (GQM) paradigm proposed by Victor Basili and H. Dieter Rombach [Basili 88]. The basic premise of the GQM paradigm is that any software measurement activity is preceded by a goal. This goal leads to questions which generally involve quantification of the goal. Quantification issues lead to measures. This module will assist the practitioner and instructor in deciding which measures are appropriate answers to which questions.

1.1. Measure Versus Metric

Many people are reluctant to use the term metric in reference to software. *The American Heritage Dictionary* (Mifflin, 1991) defines a metric as:

> 1. designating, pertaining to the metric system, or

> 2. a standard of measurement.

Mathematicians define a metric more rigorously; they use the term to apply to a real-valued set function that measures the distance (as defined by the metric) between two objects in the set. In his text on topology, Mansfield [Mansfield 63] defines a metric as follows:

Let A be a set of objects, let R be the set of real numbers, and let ρ be a one-to-one function such that $\rho : A \Box A \varnothing R$, where $\Box$ denotes the Cartesian product of A with A. Then, ρ is a metric for A if and only if

> $\Box$ $\rho(\alpha,\beta) \, \varepsilon \, 0 \ \ \forall \alpha,\beta \, \Box \, A,$

> $\Box$ $\rho(\alpha,\beta) = 0 \ \Box \ \alpha = \beta \, ,$

> $\Box$ $\rho(\alpha,\beta) = \rho(\beta,\alpha) \ \ \forall \, \alpha,\beta \, \Box \, A, \ \text{and}$

> $\Box$ $\rho(\alpha,\gamma) \, \leq \, \rho(\alpha,\beta) + \rho(\beta,\gamma) \ \ \forall \, \alpha,\beta,\gamma \, \Box \, A.$

For the purposes of this document, the term software metrics will mean measurements made on a software artifact. There are two important components of the software artifact that are measured for our purposes: the artifact's design specification document and its coded implementation.

The concept of a metric measuring the *distance* between two objects in a set A has very little meaning in the world of software. Why would we want to measure the distance between two software products or two software specifications? It does, however, make sense to measure software product X

---

and software product Y, and then, to compare the two measures. We also note that there is no standard of measurement for software artifacts that is universally accepted. Based on both the dictionary and mathematical definitions of metric, we see that the term *software metric* is not appropriate. The preferred term is *software measure*.

1.2. Standards for Measures

1.2.1. Weyuker's Measure Properties

Many issues arise as to what constitutes, and what are the acceptable properties of, a software measure. Elaine Weyuker has brought together nine properties that a software product measure should have [Weyuker 88]. Many authors have used these properties as a standard against which to evaluate their own measures.

"All the measures considered depend only on the syntactic features of the program" [Weyuker 88].

Let P, Q, and R be programs.

P + Q means that P and Q halt on the same input.

P;Q means that P is augmented by Q. (An appending of Q to P)

The measure of P is denoted by | P |.

Nine properties of measures:

1. ($\exists$ P) ($\exists$ Q) (| P | $\square$ | Q |).

2. Let c be a nonnegative number. Then there are only finitely many programs of measure c.

3. There are distinct programs P and Q such that | P | = | Q |.

4. ($\exists$ P) ($\exists$ Q) (P + Q and | P | $\square$ | Q |).

5. ($\forall$ P) ($\forall$ Q) (|P| $\leq$ | P; Q |) and (|Q| $\leq$ | P; Q |).

6. ($\exists$ P) ($\exists$ Q) ($\exists$ R) (| P | = | Q |) & (| P ; R | $\square$ | Q ; R |) and ($\exists$ P) ($\exists$ Q) ($\exists$ R) (| P | = | Q |) & (| R ; P | $\square$ | R ; Q |).

7. There are program bodies P and Q such that Q is formed by permuting the order of the statements of P; and | P | $\neq$ | Q |.

8. If P is a renaming of Q, then | P | = | Q |.

9. ($\exists$ P) ($\exists$ Q) (| P | + | Q | < | P ; Q |).

1.2.2. A Critical Analysis of Weyuker's Properties

Property number one [($\exists$ P) ($\exists$ Q) (|P| $\square$ |Q|)] reflects the idea that a measure that assigns all programs the same value is not a measure. Property number two (for a nonnegative number c there are only finitely many programs of measure c) is the non-coarseness property: it places a constraint on property one by stating that only a finite number of programs can be assigned the same measure. Property number three [there are distinct programs P and Q such that |P| = |Q|] is often called the non-uniqueness property: two different products can have the

same measure value. Property number four [($\exists$ P) ($\exists$ Q) (P + Q and |P| $\square$ |Q|)] states that two software products can possess the same functionality but not have equal measure values. Property number five [($\forall$ P) ($\forall$ Q) (|P| $\leq$ |P; Q|) and (|Q| $\leq$ |P; Q|)] is a monotonicity requirement: a combination (concatenation) of two products can never have a measure value less than either of the products taken individually. Property number six [($\exists$ P) ($\exists$ Q) ($\exists$ R) (|P| = |Q|) & (|P ; R | $\square$ |Q;R|) and ($\exists$ P) ($\exists$ Q) ($\exists$ R) (|P| = |Q|) & (|R ; P| $\square$ |R ; Q|)] states that there exist products whose measure values are the same, but the augmentation of either product by a third product can produce measure values that are not the same. Property number seven [there are program bodies P and Q such that Q is formed by permuting the order of the statements of P; and |P| $\neq$ |Q|] states that there are software products whose measure value can be affected by a permutation in the order of program statements. Property number eight [if P is a renaming of Q, then |P| = |Q|] is the "carbon copy" property indicating that the measure value is not affected by any isomorphic transformation of the original product. Property number nine [($\exists$P) ($\exists$Q) (|P| + |Q| < |P; Q|)] is the most controversial of the nine properties. This property states that augmentation increases the measure value for some software products.

Weyuker's properties are concerned with computer programs. What features of computer programs do these properties encompass? The answer to this question is unclear. Consider property number five which states "for all programs P and Q the measure of program P augmented by program Q is greater than or equal to both programs P and Q alone." This property is reasonable if the feature of concern is program size and the measure is the number of lines of executable source code. However, for the same feature program size and the same measure number of lines of executable source code, property number five is in conflict with property number six. Property six states, "there exist programs P, Q, and R such that programs P and Q can have the same measure and the measure of P augmented by R is different from Q augmented by R." This property is not true for lines of code that are used as the measure and, in fact, is not true for most size measures, suggesting that Weyuker's properties encompass some feature other than program size.

Since the title of Weyuker's article is "Evaluating Software Complexity Measures," the properties must also involve complexity. McCabe introduced a measure called the cyclomatic complexity metric $v = \pi + 1$, where π is the number of predicates in a program [McCabe 76]. A predicate in a program is a Boolean expression having one of the forms:

B1 = B2, B1 $\neq$ B2, B1 < B2, B1 > B2, B1 $\leq$ B2, or B1 $\geq$ B2,

where B1 is an identifier and B2 is either a constant or an identifier. To use the predicate count approach to compute McCabe's metric, all statements involving compound Boolean expressions are reduced to a sequence of statements with only predicates in them. Careful calculation indicates that Weyuker's property five is satisfied and property six is not satisfied. Thus, Weyuker's properties do not encompass McCabe's view of complexity.

Halstead, however, introduced a measure that does satisfy property six. The measure (called an effort measure) measures the effort involved in

producing an algorithm [Halstead 77], but the measure is difficult to compute; it involves the counts of the total occurrence of operators and operands and the counts of unique operators used and unique operands. Halstead's effort measure is implementation-dependent. Furthermore, Weyuker proves algebraically that the Halstead effort measure does not satisfy her property number five, but does satisfy her property number six.

Which features, then, of software products are encompassed by Weyuker's properties? Fenton resolves this issue by stating, "Properties five and six are relevant for very different (and incompatible) views of complexity. Hence it is impossible to define a set of axioms for a completely general view of 'complexity' [Fenton 91]." This suggests that software products have features that can be identified and grouped into categories that include features, measures, and axioms for these measures.

Weyuker's set of properties is a seminal effort in establishing a basis for evaluating software measures. Some of the properties should apply to all software measures; some apply to a chosen few features that we may wish to measure. Property number two, for example, is a property that all measures should satisfy. Simply stated, this property requires that a measure not be "too coarse." Yet, property number two is not satisfied by McCabe's cyclomatic complexity measure, in which too many programs would be assigned the same measure.

That software products have features that have conflicting properties is evidenced by established and accepted measures that do not satisfy some set of Weyuker's properties. Once a design and implementation paradigm is chosen, the features of concern of the software products to be produced should be isolated and grouped into categories. Measures can be selected for each category, and lists of properties can be developed for these measures. Weyuker's properties can be used as a basis for selecting these properties. This also suggests that a collection of measures may be appropriate for the application as opposed to a single measure.

1.3. What Can Be Measured

In *Lecture Notes on Engineering Measurement,* in the section titled Software Engineering Measures, Gary Ford comprehensively answers the question, "What can be measured?" [Ford 93]. The properties or attributes of software that are directly measurable are size, effort, schedule, and quality. These four attributes are often called the SEI core measures [Carleton 92]. Ford also observes that there are a few other software properties that are generally believed to be important, but it is not yet known how to measure them very well. Reliability, reuse, and complexity are among these properties.

In this module, the term software measure refers to the measurement of the software product and the process which produces this product. The software product can be thought of as an abstraction that evolves from a specification document into a finished software system. Specifically, a software product is considered as both the programming language source code and the design document(s). Both components of the product are seen as being measurable.

In addition, the environment in which the software product is produced influences the acceptance of the measure by the experimental community.

Measurement research has taken place in two rather distinct environments, each of which has its own unique characteristics. The result is two distinct types of experiments from which experimenters have drawn conclusions, proposed measures, and proposed models. These are large-scale and small-scale experiments. Conte, Dunsmore, and Shen were the first researchers to document these two experimental environments [Conte 86]. Accordingly, a *large-scale experiment* is an experiment that captures the characteristics of a large-scale system. These characteristics are:

A large, organized team of people, including specialists, is required to design, implement, and maintain the system.

The system is large, employing hundreds of thousands of lines of source code, hundreds of modules, and many functions to be performed.

The system reflects a variety of abilities and techniques, and is difficult for any one person to understand fully.

There are strong dependencies among system components (as opposed to a collection of independent modules).

The system's users typically did not design or write the system, yet must rely on it for accurate information.

The system must be updated often and, perhaps, several versions must be maintained simultaneously.

In contrast, a *small-scale experiment* is one involving a few subjects, usually working alone on a relatively simple task that can be completed in a matter of a few hours. A micro-model is a relationship among factors generally supported by small-scale experiments, and a macro-model is a relationship among factors generally supported by large-scale experiments. Conte and his co-authors contend, "Success in the development of micro-models may not lead to success in macro-models. However, failure in the development of micro-models can be detrimental to our confidence in macro-models" [Conte 86]. Both types of experiments and models have been reported in the literature, and both the student and instructor of software measurement should be aware of these differences and able to judge the reported finds accordingly.

1.3.1 Process

The process that takes place involving people, time, environment, tools, and management to generate the software product is measurable. Many process measures and models have been proposed in the literature. Everald Mills in his SEI document covers process measures and models [Mills 88]. Conte in his text covers process measures and models both from a micro-model viewpoint and a macro-model viewpoint [Conte 86]. Putnam covers process models from a macro-model viewpoint [Putnam 92].

The reader is reminded that process measures are not covered in this module and is referred to the three sources referenced in the paragraph above for further detail. Incidentally, (referring to the four SEI core features to be measured), effort and schedule are both directly associated with process.

1.3.2 Product

The software product is measurable. The attributes of the software product that are most commonly measured are size and quality. Product size is usually measured by lines of source code with stringent counting rules imposed. Product quality is usually measured by observed defects found and defects found per thousand lines of source code. A defect is the manifestation of a software fault. A software fault is the result of a programming error or an error in specification of the intended product. A programming error could be the result of a design error, a misinterpretation of a design specification, or simply a programming mistake. No one can guarantee the absence of faults. Some faults can be detected through design reviews, code reviews, walkthroughs, and various types of testing.

The two standard measures of defects are the count of the number of defects at some specific point in time and the number of defects per thousand lines of source code. The second measure is a ratio and is more useful to software developers; however, this ratio has a denominator, lines of source code, whose value is prone to error and inconsistent measurement. Robert Park has proposed a complete set of guidelines for counting lines of source code [Park 92]. Lawrence Putnam has collected data from large organizations and large-scale applications, and reports that defect data follow a Raleigh model [Putnam 92]. The Raleigh model stated by Putnam is

$$E_m = \left(\frac{6E_r}{t_d^2} \right) t\, e^{-3t^2/t_d^2}$$

where

E_r = total number of errors over the life of the project

E_m = errors per time period

t = time in time periods

t_d = time to develop product in time periods

NOTE: E_r is obtained from past data and adjusted proportionally for the current project.

The Raleigh model is a theoretical model whose practical use is to track the actual defect rate against the expected defect rate from the model. Excessive deviation of actual rates from expected rates at any point in time during the project is an indication of an anomaly somewhere. A significant deviation may indicate poor error detection or the presence of too many errors; both situations warrant action.

In addition to simply counting defects, additional information and insight into the source of the errors causing the defects can be gained from recording where the fault is located, when it was detected, and when it was injected. Norman Fenton devotes an entire chapter (Chapter 8) to fault-related issues in his text [Fenton 91]. Conte and his co-authors explore the financial impact of faults and, in addition, provide a mathematical derivation of the mean time to failure (MTTF) for a software component [Conte 86, pp. 93-106]. Other measures of a software

product include measures made on the design. These measures will be discussed in Section 4.

2. The Object-Oriented Paradigm

A new paradigm became popular in the mid 1980s that began to affect the way software developers viewed software analysis and design. This paradigm, the object-oriented paradigm, has compounded the study of software measures because of the multiplicity of interrelated elements. Are software products produced under object-oriented techniques measurable by existing software measures, or does a new body of measures need to be invented? What is the current state of the discipline relative to *object-oriented measures*?

2.1. Origins of the Paradigm

Ole-Johan Dahl and Kristen Nygaard of Norway created the seminal work on an object-oriented language with their introduction of Simula67 in 1967. As the name implies, Simula67 was generally used for simulation modeling and proved to be a significant influence on later object-oriented languages. Smalltalk, developed at XEROX in Palo Alto in the 1970s, was the next major development of an object-oriented language. Smalltalk was followed by a number of languages that either were object-oriented from inception, such as Eiffel, or revamped a previous language to include object-oriented capabilities, such as C++, Object Pascal, and Ada95.

An excellent treatment of the evolution of the object-oriented paradigm can be found in Grady Booch's text (Chapter 2) [Booch 94, pp. 27-72].

2.2. Features of Object-Oriented Products That Are Different from Conventional (Procedure-Oriented) Products

One feature that makes objected-oriented software products different from earlier or conventional software products is the use of procedures and subprograms. Today, the conventional technique of structured programming is procedure-oriented, but is supported by programming languages that support separate compilation of modules, data abstraction, strong data typing, and data encapsulation. That structured programming is still procedure-oriented indicates an early emphasis on implementation in the life cycle. Today and in the past, a major portion of the life cycle is spent on implementing the design.

In contrast, object-oriented programming places greater emphasis on the design phase of the software life cycle. The essence of the object-oriented design is that it decomposes the system into object classes, the basic building blocks of the object-oriented approach; gathers together the data and the functions to be performed on the data; and encapsulates the data and functions (methods) within the class.

Another feature that makes object-oriented software products different from the conventional procedure-oriented software products is the object class itself. Features of the object class (or simply class) that become measurable are the number of attributes the class contains, the number of methods the class has, the number of methods called from other classes, the number of methods outside the class that are called, and the placement of the particular

class in the class inheritance tree. Classes are complex entities and should be considered as more than a collection of methods and attributes. Classes spawn objects by a process called instantiation, and the class can no longer be thought of in a two-dimensional sense.

Emphasizing shared features of object-oriented products, abstract data types exist in conventional procedure-oriented programming languages, and classes can be implemented as abstract data types in most of the existing object-oriented languages. However, one of the key differences between the procedure-oriented implementation and the object-oriented implementation is the concept of inheritance. Inheritance is a relationship among classes in which a class shares (inherits) the attributes and methods of another class.

The methods of a class are similar to the functions, programs, or subprograms that are used in conventional programming. Functionality in classes is gained through message passing both within classes and between classes. A class's methods are measurable. Methods can be measured by the earlier, more conventional measures. Examples of such measures are lines of code, Halstead's software science metrics, McCabe's cyclomatic complexity metric, and Albrecht's function points.

Unique features of object-oriented programming and design impose added complexities on the measuring process. These features—message passing, inheritance, and polymorphism—require a suite of measures designed to handle them.

2.3. Suggested Common Terminology for Object-Oriented Approaches

In this report, we treat the term *object* as a primitive term. Objects have attributes, methods, and an identity (a name). The following terminology is a partial adaptation of Booch's set of terms [Booch 94, pp.511-520]. The author provides these definitions so that the terminology used to describe object-oriented software products in this module is as uniform as possible.

Abstraction. The essential characteristics of an object that distinguish it from all other kinds of objects, and thus provide, from the viewer's perspective, crisply-defined conceptual boundaries; the process of focusing upon the essential characteristics of an object.

Aggregate object (aggregation). An object composed of two or more other objects. An object that is *part of* two or more other objects.

Attribute. A variable or parameter that is encapsulated into an object.

Class. A set of objects that share a common data structure (called attributes) and a common behavior manifested by a set of methods; the set serves as a template from which objects can be created.[1]

Cohesion. The degree to which the methods within a class are related to one another.[2]

Collaborating classes. If a class sends a message to another class, the classes are said to be collaborating.

Coupling. Class X is coupled to class Y if and only if X sends a message to Y.

[1] There are other interpretations of class cohesion is limited to *cohesion within a class.*

[2] Here,

Depth. The depth of a class is the length of the longest path from the root of the inheritance tree to the class in question.

Encapsulation. The process of bundling together the elements of an abstraction that constitute its structure and behavior.

Information hiding. The process of hiding the structure of an a class and the implementation details of its methods. A class has a public interface and a private representation; these two elements are kept distinct.

Inheritance. A relationship among classes, wherein one class shares the structure or methods defined in one other class (for single inheritance) or in more than one other class (for multiple inheritance).

Inheritance Tree. A directed graph in which the nodes represent classes and the edges represent base-class/derived-class dependencies. The graph may not be a tree if multiple inheritance is permitted.

Instance. An object with specific structure, specific methods, and an identity.

Instantiation. The process of filling in the template of a class to produce a class from which one can create instances.

Message. A request made of one class to another, to perform an operation.

Method. An operation upon an class, defined as part of the declaration of a class.

Polymorphism. The ability of two or more objects to interpret a message differently at execution, depending upon the superclass of the calling object.

Superclass. The class from which another subclass inherits its attributes and methods.

Uses. If class X is coupled to class Y and class Y is coupled to class Z, then class X uses class Z.

It has been a common goal of many researchers to agree upon a common set of terminology that encompasses the object-oriented analysis and design methods used in the object-oriented community. Neville Churcher and Martin Sheppard recently introduced a set of terminology to achieve this goal. Two of their terms are incorporated into the set above—depth and inheritance tree [Churcher 95].

2.4. Overview of Object-Oriented Design Methods

Roger Pressman in his software engineering text summarizes object-oriented design methods by stating, "At their current stage of evolution, object-oriented design methods combine elements of all three design categories: data design, architectural design, and procedural design. By identifying classes and objects, data abstractions are created. By coupling operations to data, modules are specified and a structure for the software is established. By developing a mechanism for using the objects (e.g. generating messages), interfaces are described" [Pressman 92, p.403].

Ed Yourdon in his recent text views object-oriented design from a more general viewpoint than Pressman. Yourdon states, "So what is design—and in particular, what is object-oriented design? Fundamentally it consists of three things:

- Notation — so we can communicate our ideas about the design to other members of the project team, and to interested outsiders

- Strategies — so we don't always begin each project as if this is the first time the human race has ever considered tackling a problem of this kind, and so the designs for common domains of problems will begin to fall into familiar 'patterns' of solutions

- Goodness criteria — so we can have an objective way of evaluating a design to see if it should be accepted, rejected, or revised" [Yourdon 94, p. 250].

The primary purpose of the design is to create a framework or architecture from which the implementation will eventually evolve. The framework may take a variety of forms—graphical, narrative, or a combination of both graphics and narrative. Researchers and practitioners have developed a variety of methods, some with exotic acronyms, to produce a design framework. A few of these methods are presented here so that the reader may be aware of these approaches and the various notations that are used.

Coad-Yourdon Method.

Coad and Yourdon use the same notation for design as they do for analysis [Coad 91]. The design framework they develop is language-independent and uses numerous graphical representations. Figure 2-1 portrays a diagram for a superclass A with two subclasses (specializations); this is also known as a *is-a* hierarchy.
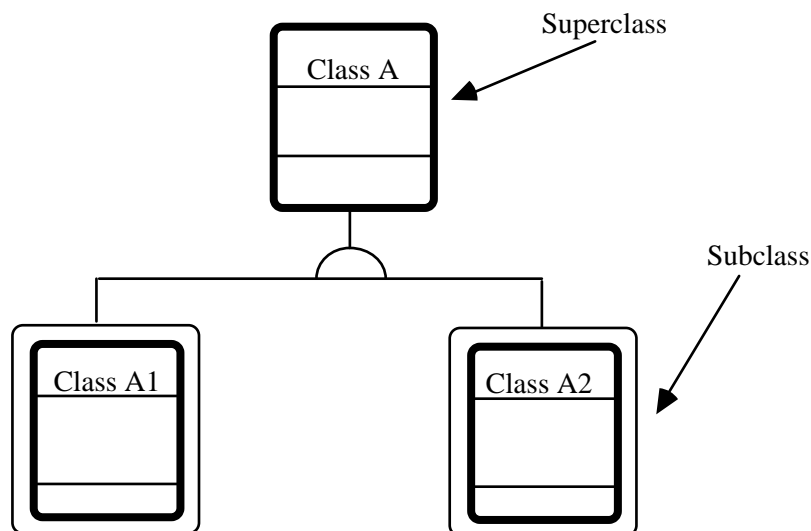
Figure 2-1: Specialization (Coad-Yourdon)

Figure 2-2 represents a whole-part (assembly) relationship; this is also known as a *has-a* relationship. Note that the notation includes bounds on the relationship. If M denotes the number of instances of PartA, then WHOLE consists of M instances of PartA where $n1 \leq M \leq n2$. If N denotes the number of instances of class WHOLE, then PartA may be a component of N instances of WHOLE where $n3 \leq N \leq n4$.
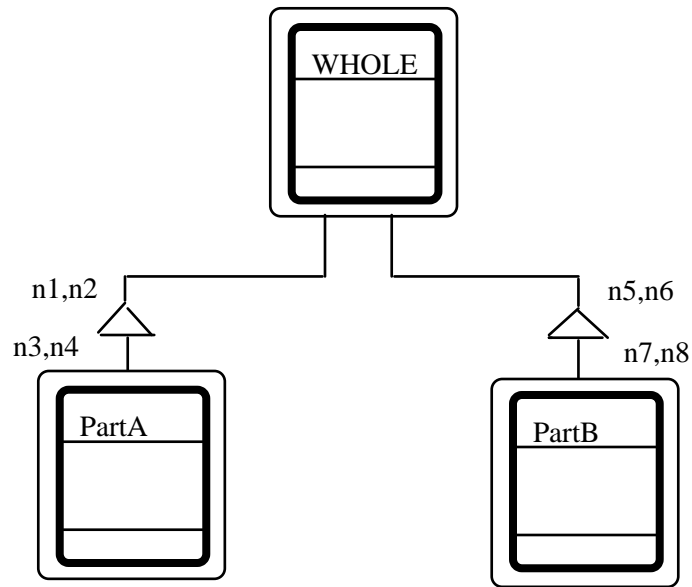
Figure 2-2: Assembly (Coad-Yourdon)

Coad and Yourdon also use object file history (usually called state-transition) diagrams and object communication diagrams (similar to the data flow diagram) to describe the behavior of object classes as they collaborate/communicate with other object classes. Figure 2-3 portrays a combination of these two diagrammatic tools. The object file history diagram for object class B suggests that the object class has three states that are in a specific order. State 1 consists of awaiting the arrival of the value of attribute X, which the object must have to produce V. State 2 consists of awaiting the arrival of the value of attribute Y, which the object must have to produce U. The final state for object class B is that of having sent both values for attributes V and U.
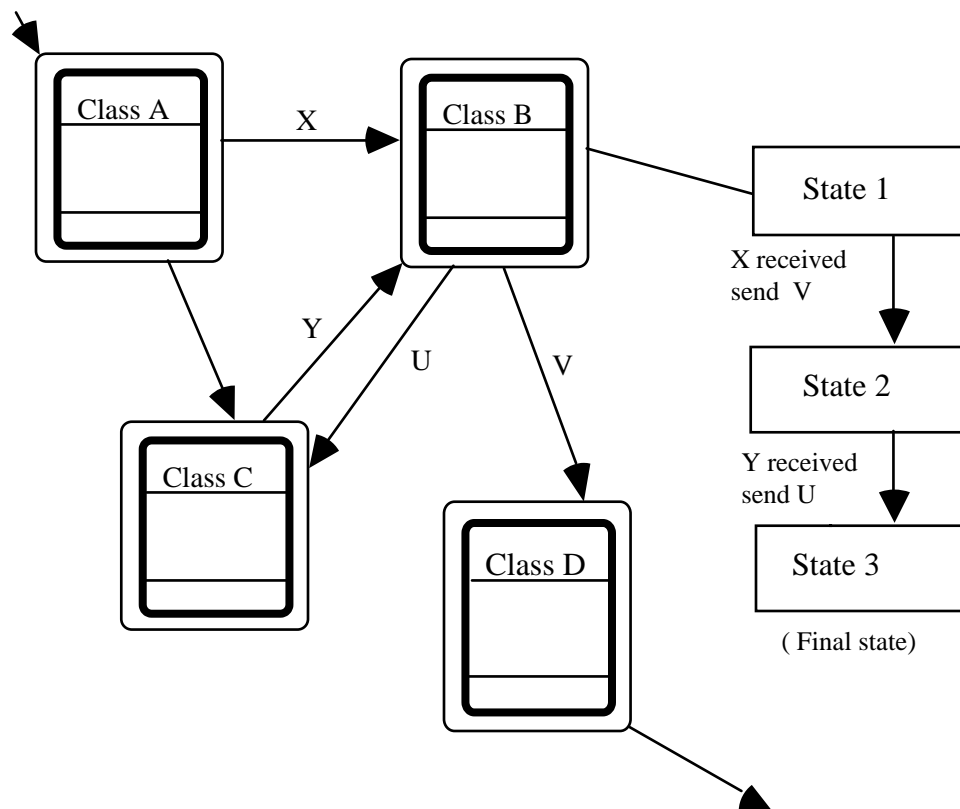
Figure 2-3: Object History and Communication (Coad-Yourdon)

Booch Method

The Booch method is rich in notation that encompasses all his design issues in some diagrammatic manor [Booch 94]. The *is-a* and *has-a* relationships are portrayed in Figures 2-4 and 2-5, respectively. Note that the class icon that Booch uses is cloud shaped.
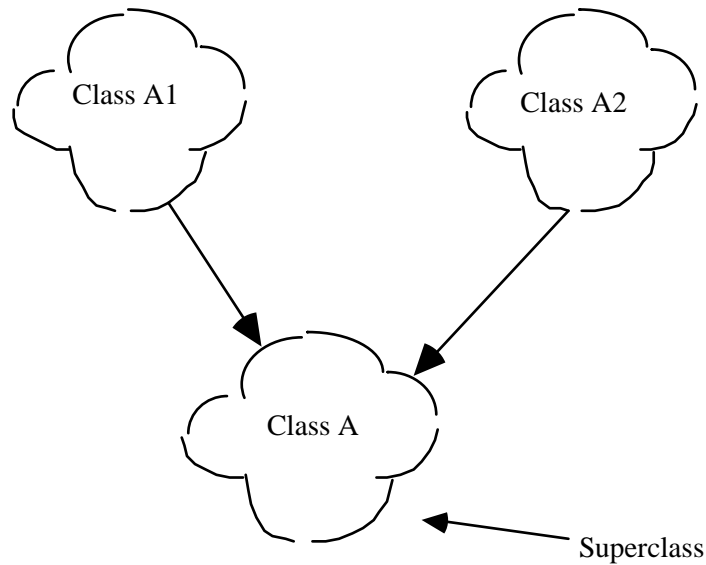
Figure 2-4: Specialization (Booch)
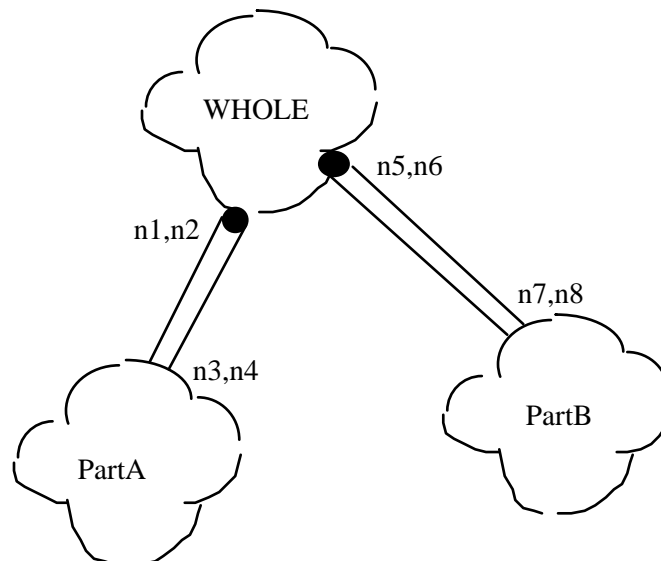
Figure 2-5: Assembly (Booch)

The numbers n1 through n8 serve the same role as in Figure 2-2.

The Firesmith Method

The Firesmith Advanced Software Technology Specialists Development Method 3 (ADM3) uses a specification and design language called OOSDL to document the design of the system. Firesmith states, "ADM3 is a third-generation, object-oriented software development method for use on large, complex, real-time projects... It has a very rich, consistent set of models and diagram classes, which can be used to model all aspects of almost all applications" [Firesmith 93, pp.231-321]. OOSDL is strongly typed, is quasi-formal, uses standard English, and is based on the Ada programming language. Figures 2-6 and 2-7 portray the *is-a* and *has-a* relationships.
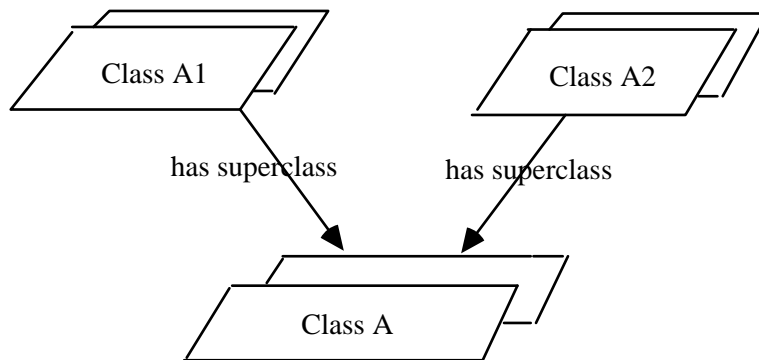
Figure 2-6: Specialization (Firesmith)

Both Firesmith and Booch use notation in which the superclass A is depicted at the bottom of the diagram, indicating the relationship as going from specialization to generalization. Firesmith uses a parallelogram to represent concurrent classes and a rectangle to represent sequential classes.
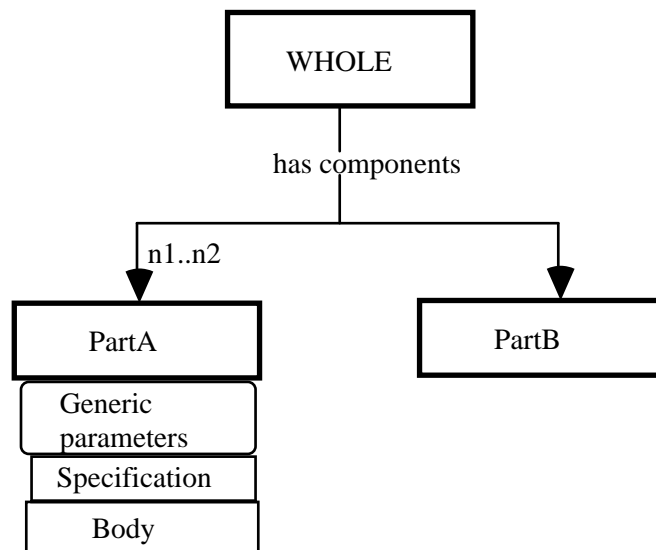
Figure 2-7: Assembly (Firesmith)

Rumbaugh and Co-Authors Method

Rumbaugh and his co-authors use the same notation for both analysis and design. According to the Rumbaugh group, "Object-oriented design is primarily a process of refinement or adding detail" [Rumbaugh 91, p.228]. One of the co-authors has written a program, Object Modeling Tool (OMTool), that is a graphic editor for constructing object diagrams. The method of Rumbaugh and co-authors uses the three analysis phase models—object, dynamic, and functional models—as a basis for attaching methods to the classes and completing the design. Chapter 10 of their text provides guidelines for designing objects and choosing algorithms. Figures 2-8 and 2-9 portray the notation for the *is-a* hierarchy (specialization) and the *has-a* relationship (assembly), which Rumbaugh calls *aggregate*.
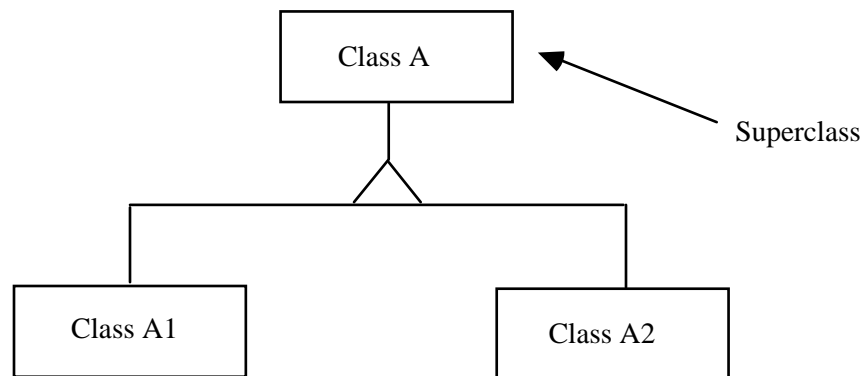
Figure 2-8: Specialization (Rumbaugh)

Rumbaugh recently joined Grady Booch's company, Rational, in 1994; so this notation and approach may change.
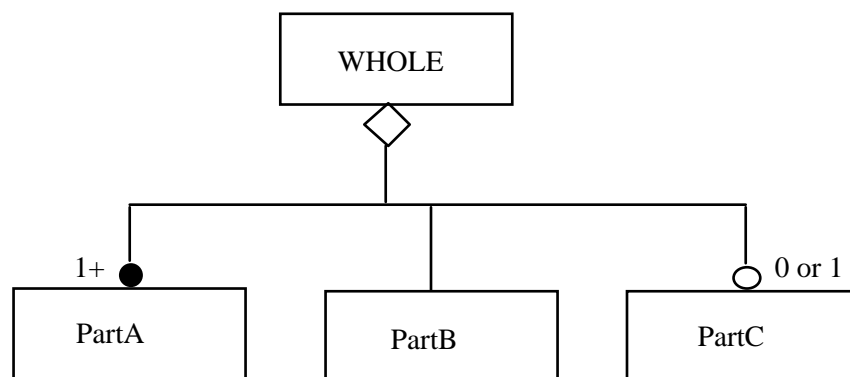
Figure 2-9: Assembly (Rumbaugh)

The Henderson-Sellers and Edwards Method

Brian Henderson-Sellers and Julian Edwards have proposed a method for object-oriented analysis and design, called MOSES (Methodology of Object oriented Software Engineering of Systems), which, in addition to design and documentation, provides a framework for project management, quality assurance, and metrics. The diagrams for specialization and assembly at the design stage are portrayed in Figures 2-10 and 2-11, respectively.
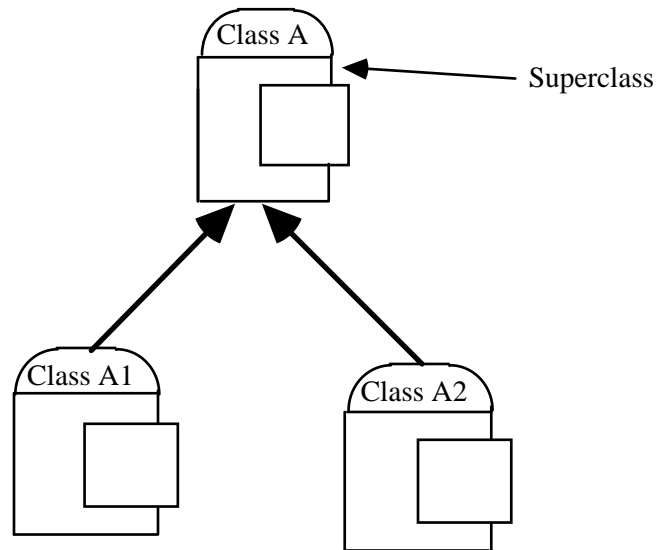
Figure 2-10: Specialization (Henderson-Sellers)

The authors use the rectangle insert in the object class icon to represent the public portion of the class at the second level of the object class diagram [Henderson-Sellers 94, pp.46-67].
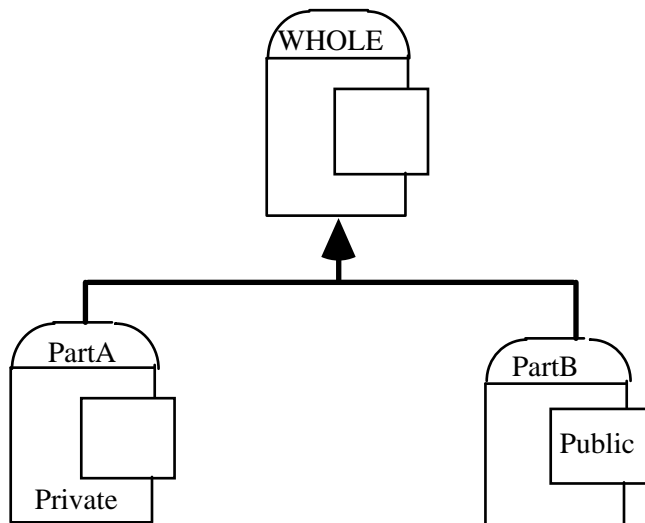
Figure 2-11: Assembly (Henderson-Sellers)

Coleman's Fusion Method

Derek Coleman and his co-authors have proposed Fusion, a method they consider to be a second-generation object-oriented software development method. According to the authors, Fusion "builds on the successful parts of earlier object-oriented methods and addresses their weaknesses. It has three phases—analysis, design and implementation" [Coleman 94, pp.11,19-22]. The notation of the method is simple and captures the essential features of the analysis and design. The diagrams for specialization and assembly are portrayed in Figures 2-12 and 2-13, respectively.
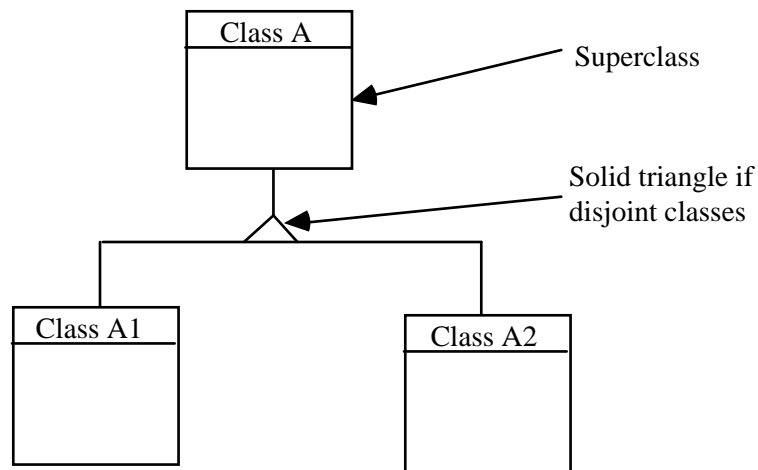
Figure 2-12: Specialization (Coleman)

For the assembly (aggregate class) Coleman denotes cardinality constraints of classes by either a single number, n1..n2 for a range, "∗" for zero or more, or "+" for one or more.
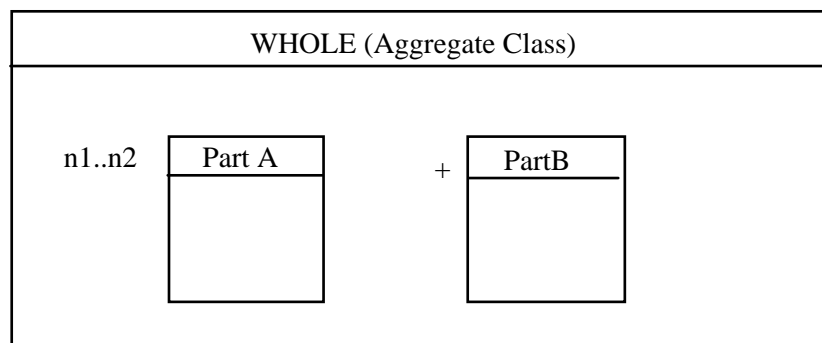
Figure 2-13: Assembly (Coleman)

The Fusion method also provides for a data dictionary to serve as a central repository of definitions of terms and concepts.

Other Methods

There are several other design methods that have also been proposed in the literature. These include the methods of Martin and Odell; Shlaer and Mellor; Wirfs-Brock, Wilkerson, and Wiener; and Ivar Jacobson and his co-authors. Of the four approaches, the text by Martin and Odell is more analysis-oriented than design-oriented, as evidenced by the fact that three of the 29 chapters specifically deal with design [Martin 92].

Shlaer and Mellor introduce a graphical diagramming notation called OODLE (Object-Oriented Design LanguagE) that uses four diagram types that are different from their analysis diagrams. OODLE is elaborate, and encompasses four key issues of design—class diagram, class structure chart, dependence diagram, and inheritance diagram [Shlaer 92, pp.201-204].

Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener have proposed yet another approach to object-oriented analysis and design. Their approach is best summarized by a quote from the preface of their book: "In this book we offer basic design principles, and a specific design process, that can be applied to any software programming effort, even those not using object-oriented languages or environments. We provide a coherent model for the design process: responsibility-driven design. We also provide tools, such as the hierarchy graph and the collaboration graph, to help the designer every step of the way." The authors have presented, possibly, the simplest icons for the object class and the various relationships—the icon for an object class is a rectangle. Their text has a solid chapter on advice for implementing a design [Wirfs-Brock 90, pp.177-190].

Ivar Jacobson and his co-authors propose Objectory (the *Object* Fact*ory* for Software Development) as a development technique, which has the properties that it must support the iterative development of a system over the entire life cycle, it should view each iteration as a change to an existing system, and it must support the entire chain from changed requirements to the functioning system [Jacobson 92, pp.39-40]. The nucleus of Objectory is the "use case," which is a scenario that some part of the system must perform. This technique has been used successfully in Sweden by the telecommunications industry.

3. Features of Object-Oriented Software Products

3.1. A Suggested Taxonomy for Features

The object-oriented design approach gives rise to a natural taxonomy that incorporates the salient features and properties of an object-oriented software product. Archer and Stinson have proposed such a taxonomy [Archer 95, p 13]. This taxonomy captures these properties hierarchically. It begins with the high-level characteristics of an object-oriented system and moves down to the low-level characteristics.

System. The system and its components are at the highest level. Although a system can be subdivided into components, these components also act as a system. Also, the characteristics of a good component are those of a good system and vice versa. The measurable characteristics of a system might include the number of classes in the system or the number of edges in the inheritance tree(s) for the system.

Coupling and Uses. Classes often interact with other classes to form a subsystem. Characteristics of this interaction may indicate a complexity resulting from too much coupling, or from using objects derived from objects that have been obtained from yet another object. Such complexity can complicate the programming process. Uses and coupling are related issues; uses is defined in terms of coupling. The role of uses and coupling in the interaction of classes makes them a single taxon: both capture the interaction of classes.

Inheritance. Classes are found in a class structure diagram, often called an inheritance tree or class hierarchy graph (the graph may not be a tree if multiple inheritance is permitted). Visible in the graph are the inheritance relationships between classes and their parents—the properties shared by both. Such relationships may indicate to a designer where changes would improve the development. The inheritance tree itself contains interesting characteristics, such as the depth and breadth of the tree.

Class. The class is the main building block of an object-oriented design. Classes allow us to describe in one central location the state and the generic behavior of a set of objects, and instantiate objects that exhibit this behavior whenever we need them. Classes have many characteristics that are measurable and may have characteristics that make them excellent candidates for inclusion in a library for reuse. Classes, most certainly, deserve to be a separate taxon.

Method. Attributes and methods occur at the finest level of detail in the class. Methods are usually implemented much like procedures are in structured programming. However, in object-oriented products methods have the additional complexity of message passing. Messages can be passed to objects in the same class or to objects in different classes. Message passing involves accessing features of other objects that are visible (public) and some that are invisible (private) to the object. Such accesses should be measured or recorded.

This taxonomy attempts to encompass all the characteristics of object-oriented software products and to capture the features of the design from the system level down to the class level. These taxa also provide insight into potential areas of concern, such as depth of inheritance, cohesion, coupling, size of classes, and system structure.

Other taxonomies have been proposed. Fernando Abreu and Rogério Carapuça provide a taxonomy for measures of both object-oriented products and processes. This taxonomy, TAPROOT, deals with both product and process measures. The author's taxonomy is based on a Cartesian product of the two vectors: (design, size, complexity, reuse, productivity, quality) and (method, class, system). This produces 18 possible cells into which a metric can reside. Class and system quality measures that the authors suggest are based on counts of observed defects, failures, and time between failures. TAPROOT is presented as a starting point from which further refinement and verification can follow [Abreu 94].

3.2 Existing Measures by Taxon

3.2.1 System Measures

- SC1 - System Complexity (total length of inheritance chain) [Abreu 94]
- SR1 - System Reuse (% reused "as is" classes) [Abreu 94]
- SR2 - System Reuse (% reused classes with adaptation) [Abreu 94]
- SR3 - System Reuse (library quality factor) [Abreu 94]
- OC - Object Counts (count of object instances in the system) [Banker 91]
- OP - Object Points (count of the weighted instances of an object) [Banker 91]
- RL - Reuse Leverage (ratio of total of objects in the system to the number of unique objects built for the system) [Banker 91]
- Size - Size of Object-Oriented system (a statistical estimate based on a model developed by the author) [Laranjeira 90]
- HC - Hierarchy Complexity of system [Lee 93, p.306]
- PC - Program Complexity (defined as the sum of the complexity of the main program and the complexity of the class hierarchies in the system) [Lee 93 p.307]
- CBC - Count of Base Classes [Williams 93]
- CSC - Count of Standalone Classes [Williams 93]

3.2.2 Coupling and Uses Measures

Coupling has been defined loosely by some authors and appears to be a source of confusion to others. One definition that appears in the literature defines coupling as "a measure of interconnection among modules in a software structure" [Pressman 92, p.336]. But coupling is not a measure; coupling is a property or attribute of a pair of software modules. Coupling can be characterized as a binary relation defined on pairs of software modules. If this relation is denoted by R, then $(x,y) \in R$ if and only if some property involving x and y is satisfied. Clearly, $(x,y) \neq (y,x)$ for a general relation; that is, 'x is coupled to y' is not the same as 'y is coupled to x.' The more reasonable approach to coupling is to treat it as a binary relation on pairs of software modules (objects). The degree of coupling can be assigned a measure as some of the authors below have done.

- OCM - Operation Coupling Metric (a count of the number of operations that access other classes, are accessed by other classes, and cooperate with other classes) [Chen 93, p.234]
- CCM - Class Coupling Metric (counts accesses between classes; author explains difference between OCM and CCM by example) [Chen 93]
- CBO - Coupling Between Object classes (count of coupling, where coupling is considered as bi-directional) [Chidamber 94, p.486]
- MPC - Message Passing Coupling (number of send statements in a class) [Li 93]
- GSDM - Graph of Source and Destination of Messages (no measure given, author proposes a diagram) [Moreau 90a]

- NOT - Number Of Tramps (count of extraneous parameters that are not involved in any message passing) [Sharble 93]
- VOD - Violations Of the law of Demeter (see Lieberherr [Lieberherr 89] to understand the concept) [Sharble 93]
- COU - Count Of Uses [Williams 93]
- CCR - Count of number of Contains Relationships (This is not explained in the article; inheritance is probably the relationship measured.) [Williams 93]

3.2.3 Inheritance Measures

- AID - Average Inheritance Depth [Yap 93a]
- CHM - Class Hierarchy Metric (This is defined by Chen as the summation of a specific class in the inheritance tree, the number of subclasses of the class, the number of 'direct' superclasses of the class, and the number of local or inherited operations available to the class [Chen 93]. Chen gives an incomplete example of this measure in his paper. This measure could also be classified as a class measure.)
- DIT - Depth of Inheritance Tree [Chidamber 94]
- NOC - Number Of Children [Chidamber 94]
- IL - Inheritance Lattice (stated as being measurable, but no measure was given) [Moreau 90ab]

3.2.4 Class Measures

- CC2 - Class Complexity (progeny count) [Abreu 94]
- CC3 - Class Complexity (parent count) [Abreu 94]
- CR1 - Class Reuse (% of inherited methods that are overloaded) [Abreu 94]
- CR2 - Class Reuse (number of times class is reused "as is") [Abreu 94]
- CR3 - Class Reuse (number of times class is reused with adaptation) [Abreu 94]
- RFC1 - Raw Function Counts (represents a simple count of the five function types from function points analysis) [Banker 91]
- OXM - Operation Complexity Metric (within a class) [Chen 93]
- OACM - Operation Argument Complexity Metric [Chen 93]
- ACM - Attribute Complexity Metric [Chen 93]
- RFC2 - Response For a Class (the cardinality of the set of all methods that can potentially be executed in response to a message received by an object of the class being measured) [Chidamber 94]
- LCOM - Lack of Cohesion Of Methods [Chidamber 94]
- WMC - Weighted Methods per Class (this is simply the sum of the complexities of the methods in a class) [Chidamber 94, p.481]
- CC - Class Complexity (Lee considers a class as a collection of methods and, thus, the complexity of a class is the sum of the individual method complexities.) [Lee 93, p.305]
- DAC - Data Abstraction Coupling (number of abstract data types) [Li 93]

- NOM - Number Of local Methods (count of number of methods in a class) [Li 93]
- Size2 - number of attributes + number of local methods [Li 93]
- WAC - Weighted Attributes per Class [Sharble 93]

3.2.5 Method Measures

Neville Churcher and Martin Shepperd suggest that, in modeling the object-oriented architecture, it is appropriate to use a traditional model such as the entity-relationship model. They also state, "The relational model is sufficiently standard that, despite its limitations, it provides a sound platform for development of portable models and tools" [Churcher 95]. Churcher and Shepperd also provide a table that isolates some object-oriented software product features that are important to represent; these are: class, method, variable, and message. These one-dimensional features have been captured in the proposed taxonomy above and are also represented in the various measures that have been proposed. However, some of the binary (two-dimensional) relationships between the features are not represented by any of the measures proposed as of yet. The practitioner and instructor need to examine the Churcher and Shepperd viewpoint and evaluate these issues in terms of their own environment and needs.

- FP - Function Points [Albrecht 79]. Function points have received much attention concerning their applicability to object-oriented software products. Banker and his co-authors have shown that function points do not apply to CASE-generated code [Banker 91]; Charles Symons reinforces the weaknesses of FP counts and proposes an improvement [Symons 88]; Capers Jones has shown a positive relationship between LOC and FP count and suggests that FP can replace LOC in the traditional cost estimation models. [Jones 86].
- SSM - Software Science Metrics [Halstead 77]. There are several excellent sources that cover the software science measures; these are [Fenton 91, pp.52-54], [Conte 86, pp.37-42], and [Mills 88, p. 9].
- MCC - McCabe's Cyclomatic Complexity metric [McCabe 76]. This measure has also received much attention and is covered in the following sources: [Fenton 91, pp.181, 219-221], [Conte 86, pp.66-70], and [Mills 88, p.8].
- MC - Method Complexity. Lee has a complicated formula for computing the complexity. It involves length of method, number of arguments, coupling to other methods, etc. [Lee 93. p.304]
- Size1 - number of semi-colons in a class [Li 93]
- LOC - Lines Of executable source Code. There are many interpretations of this measure. Park is one of the best sources of guidelines for collecting this data [Park 92].

3.3 Summary of Measures for Each Taxon

Table 3-1 provides a summary of the measures given for the five taxa. The table contains the taxa names by column and the authors by row. (This table contains all the measures proposed in the literature known to me at the time this module went to publication.) Ten method measures are cited in the column; however, only six of these measures (FP, SSM, MCC, MC, Size1, and LOC) are unique (this explains the column total being written as 10{6}).

Table 3-1: Measures by Taxon

Author	System	Coupling & Uses	Inheri- tance	Class	Method
Abreu 94	SC1 SR1 SR2 SR3			CC2 CR1 CC3 CR2 CR3	
Banker 91	OC OP RL			RFC1	FP
Chen 93		CCM OCM	CHM	OXM RM OACM ACM CM	
Chidamber 94		CBO	DIT NOC	WMC RFC2 LOCM	
Coppick 92					SSM MCC
Laranjeira 90	Size				
Lee 93	HC PC			CC	MC
Li 93		MPC		DAC Size2 NOM	Size1
Moreau 90ab		GSDM	IL		SSM MCC
Sharble 93		NOT VOD		WAC	
Tegarden 92					SSM MCC LOC
Williams 93	CBC CSC	CCR COU			
Yap 93a			AID		
Total Measures	12	9	5	19	10 {6}

4. Examples of Object-Oriented Software Product Measures

4.1 Selection of Measures Suite

As indicated in Section 3, many measures have been presented in the literature on object-oriented software product measures. As an example of a measures suite, I have chosen a suite of six measures, one representing each of the taxa presented in Section 3.1, plus one more measure from the *method* taxon. These measures are listed in Table 4-1 below.

Table 4-1: Representative Measures

Taxon	Measure Chosen for Taxon	Description of Measure	Reference
Method	MCC	McCabe's Cyclomatic Complexity metric	[Tegarden 92] & [McCabe 76]
	LOC	Lines Of Code	[Park 92]
Class	Size2	Total number of attributes and methods for a class	[Li 93]
Inheritance	DIT	The depth of the inheritance tree	[Chidamber 94]
Coupling and Uses	CCM	The summation of the number of accesses to other classes, the accesses by other classes, and the number of 'co-operating' classes.	[Chen 93]
System	SC1	The total number of edges in the hierarchy graph for the system.	[Abreu 94]

4.2. C++ Example (Computer Performance)

This example is based on the sample application program taken from the Johnsonbaugh and Kalin text on object-oriented programming in C++ Johnsonbaugh, (Richard & Kalin, Martin. *Object-Oriented Programming in C++*. Englewood Cliffs, NJ: Prentice-Hall, 1995).

The C++ implementation code for the methods is omitted, some documentation has been added, and the #include statements are not included. This example is not intended to be executable, but to emphasize the computation of the various measures that apply to an object-oriented software product.

This example, shown in Figure 4-1, is the design and top-level implementation of a software artifact to simulate the measurement of computer performance. The design consists of creating two base classes, BenchMark and Computer. Class BenchMark has JobA, JobB, and JobC as derived classes. Class Computer has DeskTop and Mainframe as derived classes, and DeskTop has WS (WorkStation) and PC (Personal Computer) as derived classes. A program, TestIt, simulates a computer "running" a benchmark and outputting the results of the test. The relationships between the base classes and the derived classes are evident in Figure 4-1.
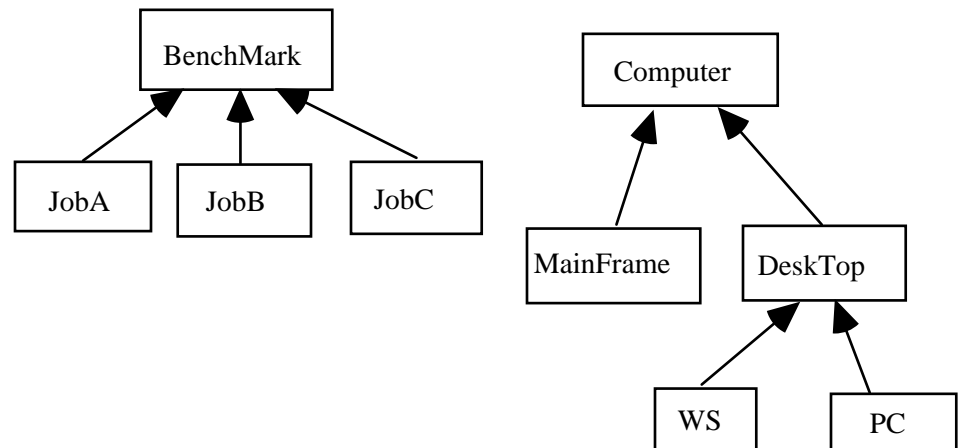
Figure 4-1: C++ Example Class Inheritance Tree

A partially coded implementation of the Computer Performance example follows:

```
      const int MaxName = 100; const float Tolerance = 0.01;
      class Test;
      class BenchMark  {
      friend Test;
      protected:
        //    Computer  instructions  are  broken  down  into
      categories
        //   percentage [expressed as a decimal, 50% = 0.50]
        float  alP; //  Arithmetic/logic instructions
        float  mP;  //  Memory
        float  cP;  //  Control instruction
        float  ioP; //  Input/output instruction
        float  ic;  //  Executed instruction count
        char   name[ MaxName + 1 ];
      public;
        BenchMark()  //  base class constructor
        {
              init();
              strcopy( name, "?????" );
        }

        BenchMark( char* n )
        {
              init();
              if (strlen(n) < MaxName)
                     strcpy(name, n);
              else
                     strncpy(name, n, MaxName);
        }      //   MCC = 2, LOC = 8

        void report()
        {
              cout << "Benchmark "<< name << endl;
              cout << " Tot ins executed == "<< ic << endl;
              cout << " A/L == "<< alP << endl;
              cout << " Memory == "<< mP << endl;
              cout << " Control == "<< cP << endl;
              cout << " I/O == "<< ioP << endl;
        }      //   MCC = 1, LOC = 9

        int okay()  // Checks to see if instruction %s sum
                    //    within tolerance to 1.0
        {
              return  fabs(1.0  -(alP  +  mP  +  cP  +  ioP))  <=
      Tolerance;
        }      //   MCC = 2, LOC = 4

        void init_error() // Print error message when invkd
        {
              //    single cout statement
        }      //   MCC = 1, LOC = 4
      private:
        void init()       // Initialize percentages to  0.0
        {
              alP = cP = mP = ioP = ic = 0.0;
        }
      };       //   ===  end of class BenchMark  ===

      class JobA :  public BenchMark {
      //  This instantiation emphasizes arithmetic/logic and
      //  control statements with moderate memory use and
      //  low I/O.
      public:
        JobA() : BenchMark( "Job  A" ) // JobA constructor
```

```
    {
          alP = 0.50; cP = 0.20;
          mP = 0.20;  ioP = 0.10;
          ic = ( float ) 4500301;
          if ( !okay() )  init_error;
    }
};       //   ===  end of class JobA  ===

class JobB :  public BenchMark {
//  This instantiation emphasizes arithmetic/logic and
//   control statements with light memory use and
//   no I/O.
public:
  JobB() : BenchMark( "Job  B" ) //   JobB constructor
    {
          alP = 0.77; cP = 0.166;
          mP = 0.064; ioP = 0.0;
          ic = ( float ) 6700909;
          if ( !okay() )  init_error;
    }
};       //   ===  end of class JobB  ===
class JobC :  public BenchMark {
//  This instantiation emphasizes low arithmetic/logic
//   andcontrol statements with heavy memory use &
//   moderate I/O.
public:
  JobC() : BenchMark( "Job  C" ) //   JobC constructor
    {
          alP = 0.153;        cP = 0.0059;
          mP = 0.577; ioP = 0.26;
          ic = ( float ) 10400500;
          if ( !okay() )  init_error;
    }
};       //   ===  end of class JobC  ===
```

```
class  Computer  {
friend  TestIt;
protected:
  //   cpi  =  cycles per instruction
  float  alcpi;      //  Arithmetic/logic cpi
  float  ccpi;       //  Control cpi
  float  mcpi;       //  Memory cpi
  float  iocpi;      //  Input/output cpi
  float  ct;  //  Cycle time in nanoseconds
  char name [ MaxName + 1 ];
  float costU; // Upper bound of cost range in dollars
  float costL; // Lower bound of cost range in dollars

protected:
  Computer( float al,  float c,  float m,  float io,
        float t,  char* n,  float lbd,  float ubd )
  {
        alcpi = al; ccpi = c;   iocpi = io;
        mcpi = m;   ct = t;
        if  ( strlen(n), MaxName )
              strcpy( name, n );
        else
              strncpy( name, n, MaxName );
        costU = ubd;        costL = lbd;
  }       //   MCC = 2, LOC = 11

  void report ()
  {
        //  cout statements to print cost range, time,
        //   and cpi values
  }       //  MCC = 1, LOC = 4
};       //  ===  end of class  Computer  ===

class Desktop:  public Computer  {
protected:
  Desktop(    float  al,  //  Arithmetic/logic
              float  c,   //  Control
              float  m,   //  Memory
              float  io,  //  Input/output
              float  t,   //  Cycle time in nanosec
              char*  n,   //  Name
              float  l,   //  Lower bd of cost range
              float  u )  //  Upper bd of cost range
        :  Computer( al, c, m, io, t, n, l, u )  {}
};       //   ===  end of class  Desktop  ===
```

```
class PC :  public  Desktop  { //  Personal Computer
public:
  PC (  float  al = 1.8,  //  Arithmetic/logic
        float   c = 2.3,  //  Control
        float   m = 5.6,  //  Memory
        float  io = 9.2,  //  Input/output
        float   t = 230.0, // Cycle time in nanosec
        char*  n = "PC",  //  Name
        float  l = 800.0, //   Lower bd of cost range
        float  u = 14500.0) //  Upper bd of cost range
        :  Desktop ( al, c, m, io, t, n, l, u )  {}
};      //   ===  end of class  PC   ===

class WS :  public  Desktop  { //  Workstation
public:
  WS (  float  al = 1.3,  //  Arithmetic/logic
        float   c = 1.7,  //  Control
        float   m = 2.1,  //  Memory
        float  io = 5.8,  //  Input/output
        float   t = 90.0, //  Cycle time in nanosec
        char*  n = "WS",  //  Name
        float  l = 4500.0, //  Lower bd of cost range
        float  u = 78900.0)  // Upper bd of cost range
        :  Desktop ( al, c, m, io, t, n, l, u )  {}
};      //   ===  end of class  WS   ===

class Mainframe :  public  Computer  {  // Mainframe
public:
  Mainframe ( float  al = 1.2, //  Arithmetic/logic
        float   c = 1.5,  //  Control
        float   m = 3.6,  //  Memory
        float  io = 3.2,  //  Input/output
        float   t = 50.0, //  Cycle time in nanosec
        char*   n = "$$$",    //  Name
        float   l = 310000.0, //  Lower bd cost range
        float  u = 20000000.0) // Upper bd of cost
                               // range
        :  Computer( al, c, m, io, t, n, l, u )  {}
};
  //   ===  end of class  Mainframe  ===
```

```
class TestIt  {
  //    Computes  the  response  time  in  nanoseconds  of
running benchmark b
  //     on computer c,  where
  //           rt  =  response time,
  //           ct  =  clock cycle time,
  //           ic  =  instruction count, and
  //          cpi  =  clock cycles per instruction.
  //    The response time in nanoseconds is computed
  //    as     rt  =  ic * cpi * ct .
  float rt;
  void results ( Computer c,  BenchMark b );
public:
  TestIt ( Computer c, BenchMark b );
};

int TestIt :: TestIT( Computer c, BenchMark b )
{
  float  al_rt,  c_rt,  m_rt,  io_rt;
  al_rt = b.alP * b.ic * c.alcpi * c.ct;
  c_rt = b.cP * b.ic * c.ccpi * c.ct;
  m_rt = b.mP * b.ic * c.mcpi * c.ct;
  io_rt = b.ioP * b.ic * c.iocpi * c.ct;
  rt = al_rt + c_rt + m_rt + io_rt;
  results (c, b);
}

void TestIt :: results ( Computer c, BenchMark b )
{
  //  cout  statements denoting computer and benchmark
  b.report();
  c.report();
} //  MCC  =  1
//     ===  End of Class TestIt  ===
//     =====  End of Example  =====
```

4.2.1 Computation of Measures for C++ Example

The computer performance example in Figure 4-2 has two base classes, BenchMark and Computer.
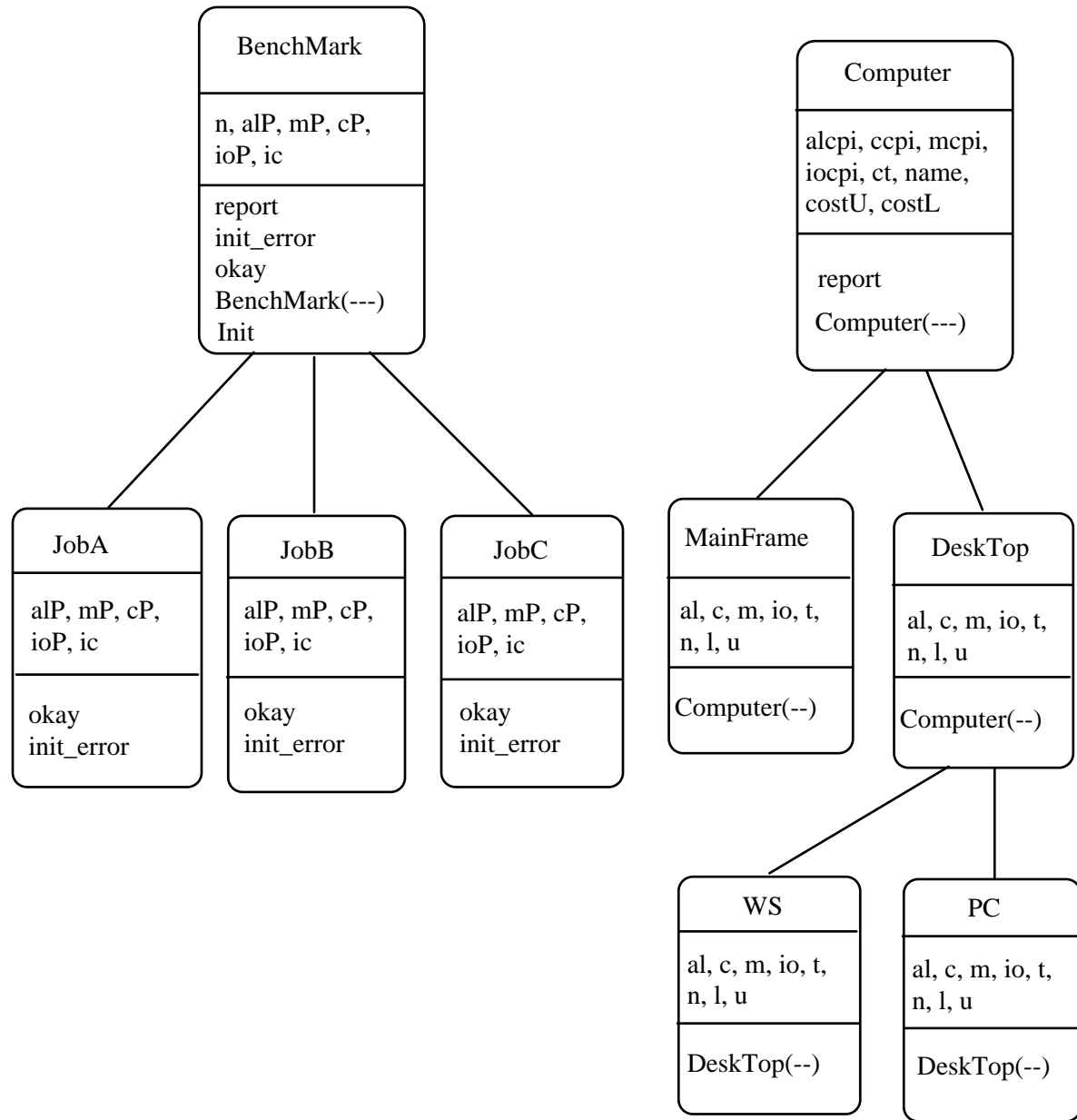
Figure 4-2: C++ Example Class Diagram

For the taxon *method,* McCabe's cyclomatic complexity (MCC) and lines of code are calculated for each method in each of the classes. The base class BenchMark has five methods (four public and one private), which are inherited by three objects formed from this base class; so we need only consider these five methods for the base class BenchMark. Similarly, base class Computer has two methods, which are inherited by two objects formed from this base class; so we need only consider these two methods for the base class Computer. TestIt is a program that accesses the two classes to instantiate the objects. The results of the calculation of MCC and LOC are summarized below. Note that MCC has values of only 1 and 2, hence not providing much granularity. Choosing both a complexity measure and a size measure for the taxon method is a better choice. Careful examination of the measures show that the two methods, okay and Computer, are 'more complex' methods and method okay has a greater density of decision structures per lines of code than does Computer.

Class	Method	Measures
BenchMark	report	MCC = 1, LOC = 8
	init_error	MCC = 1, LOC = 9
	okay	MCC = 2, LOC = 4
	Benchmark	MCC = 1, LOC = 4
	Init	MCC = 1, LOC = 4
Computer	report	MCC = 1, LOC= 4
	Computer	MCC = 2, LOC= 11

For the taxon *class,* the measure Size2 proposed by Li and Henry [Li 93] is calculated for each class in the software system. Recall that Size2 is the total number of attributes and methods for each class. The results of the calculation of Size2 are summarized below.

Class	Measure
BenchMark	Size2 = 6+4 = 10
Computer	Size2 = 8+2 = 10

For the taxon *inheritance,* the measure is DIT (Depth of Inheritance Tree) proposed by Chidamber and Kemerer [Chidamber 94]. The results of the calculation of DIT are summarized below.

Class	Measure
BenchMark	DIT = 1
Computer	DIT = 2

For the taxon *coupling and uses,* the measure is CCM (total Count of the number of accesses to other Classes, accesses by other classes and the nuMber of cooperating classes) proposed by Chen [Chen 93]. The results of the calculation of CCM are summarized below.

JobA, JobB, & JobC access BenchMark
count is 3

DeskTop and Mainframe access Computer
count is 2

PC and WS access DeskTop
count is 2

Computer and BenchMark are accessed by TestIt
count is 2

CCM = 9

For the taxon *system,* the measure is SC1 (the total number of edges in the hierarchy graph for the system) proposed by Abreu and Carapuça [Abreu 94]. From Figure 4-2, the total number of edges in the hierarchy graph for the system is seven. (Simply count the arrow heads.)

SC1 = 7

4.3. Ada95 Example (Car Dashboard Instrumentation)

This example is based on the sample application program provided by the New York University GNU Ada Translator system (GNAT) [Schonberg 94]. The Ada95 implementation code for the methods is omitted, some documentation has been added, and the with and use statements are not included. This example is not intended to be executable, but to emphasize the computation of the various measures that apply to an object-oriented software product.

This example, the hierarchy of which is portrayed in Figure 4-3, is the design and top-level implementation of a software artifact to simulate some of the instruments on an automobile dashboard. The design consists of a base class, Instrument; and derived classes, Speedometer, Gauge, and Clock.
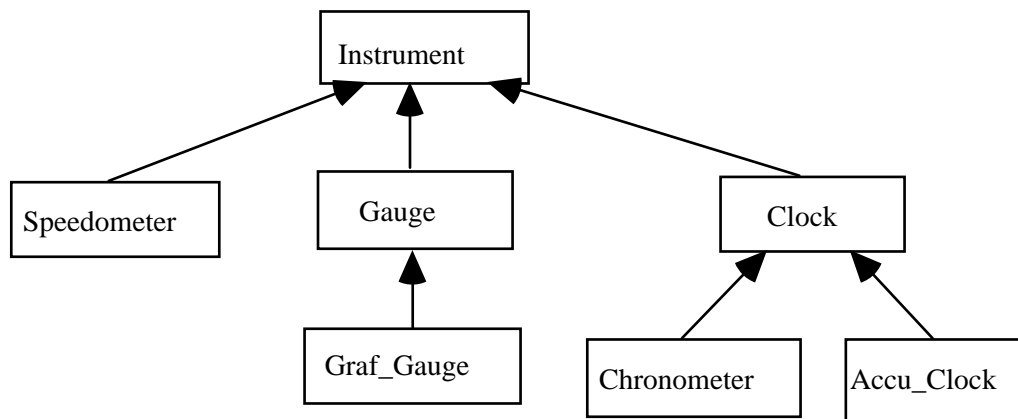
Figure 4-3: Ada95 Example Class Hierarchy Chart

A partially coded implementation of the Car Dashboard Instrumentation example follows:

```
package Instruments is
  ------------------------
  --      Root Type     ---
  ------------------------
  type Instrument is tagged record
       Name : String (1..14) := "              ";
  end record;
  procedure Set_Name (I:  in  out  Instrument;  S:
       String);
  procedure Display_Value (I: Instrument);


  ------------------------
  --   Speedometer      ---
  ------------------------
  subtype Speed is Integer range 0..85; --mph
  type Speedometer is new Instrument with record
       Value : Speed;
  end record;
  procedure Display_Value (S : Speedometer);


  ------------------------
  --      Gauges        ---
  ------------------------
  subtype Percent is Integer range 0..100;
  type Gauge is new Instrument with record;
       Value : Percent;
  end record;
  procedure Display_Value (G: Gauge);


  type Graf_Gauge is new Gauge with record
       Size : Integer  := 20;
       Full : Character  := '*';
       Empty: Character  := '.';
  end record;
  procedure Display_Value (G: Graf_Gauge);
  ------------------------
  --      Clocks        ---
  ------------------------
  subtype Sixty is Integer range 0..59;
  subtype Twenty_Four is Integer range 0..23;
  type Clock is new Instrument with record
       Seconds : Sixty  := 0;
       Minutes : Sixty  := 0;
       Hours   : Twenty_Four  := 0;
  end record;
```

```
      procedure Display_Value (C: Clock):
      procedure Init (C: in out Clock;
            H: Twenty_Four := 0;
            M, S: Sixty := 0);
      procedure Increment (C:in out Clock;
      Inc:Integer :=1);

      type Chronometer is new Clock with null record;
      procedure Display_Value (C: Chronometer);

      subtype Thousand is Integer range 0..999;
      type Accu_Clock is new Clock with record
            MSecs : Thousand  = 0;
      end record;
      procedure Display_Value (C: Acc_Clock);
    end Instruments;  -- End Class Instruments --



    --------------------------------------------
    --  Program to test the Class Instrument --
    --------------------------------------------
     procedure Test_Instruments is
     type ACC is access all Instrument'Class;
     package  DashBoard  is  new  Gen_List(Acc);  use
    DashBoard;

     procedure Print_DashBoard (L: List) is
            L1 : List  := L;
            A  : Acc;
     begin
            while L1 /= Nil loop
              A := Element(L1);
              Display_Value(A.all);
              L1 := Tail(L1);
              end loop;
            New_Line;
     end Print_DashBoard;
```

```
            --  >>>  Objects  <<<  --
Speed  : aliased Speedometer;
Fuel   : aliased Gauge;
Oil, Water : aliased Graf_Gauge;
Time   : aliased Clock;
Chrono : aliased Chronometer;
DB     : List  := Nil;
begin
      Set_Name (Speed, "Current speed");
      Set_Name (Fuel , "Fuel tank");
      Set_Name (Water, "Water ");
      Set_Name (Oil  , "Oil ");
      Set_Name (Time, "Current time");
      Set_Name (Chrono, "Chronometer");
      Speed.Value := 45;  --mph
      Fuel.Value  := 60;  --%
      Water.Value := 80;  --%
      Oil.Value   := 30;  --%
      Init (Time, 12, 15, 00);  --hrs, mins, sec
      Init (Chrono, 22, 12, 56);
      DB := Append (Speed'Access, Append
            (Fuel'Access, Append (Water'Access,
            Append (Oil'Access, Append
            (Time'Access, Chrono'Access)))));
      Print_DashBoard (DB);
end Test_Instruments;
```

4.3.1 Computation of Measures for Ada95 Example

This example, whose hierarchy chart is portrayed in Figure 4-4, has base class Instrument having three derived classes. The program Test_Instruments instantiates the objects to simulate the functions of the instrument panel.
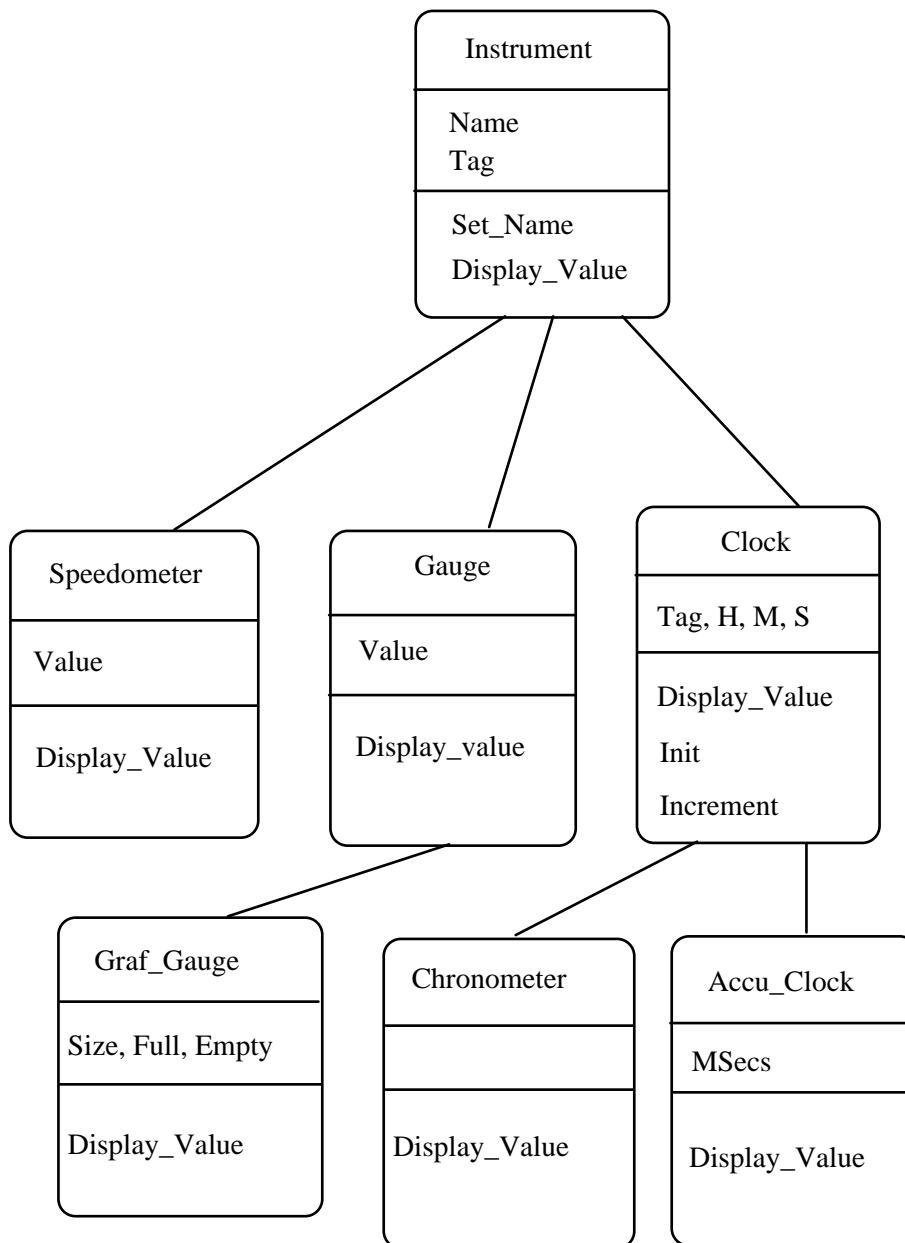
Figure 4-4: Ada95 Example Class Diagram

For the taxon *method,* McCabe's cyclomatic complexity (MCC) and lines of code (LOC) are calculated for each method in each of the classes. In Figure 4-4, we observe that the base class Instrument has two methods and the derived classes, Speedometer and Gauge, inherit these methods; so we need only consider the two methods. The derived class Gauge has a child class Graf_Gauge, which adds no new methods. Derived class Clock inherits Display_Value and adds two new methods, Init and Increment. Test_Instrument is a program which accesses the base classes to instantiate the objects. The results of the calculation of MCC and LOC are summarized below.

Class	Method	Measure
Instrument	Set_Name	MCC = 1, LOC=4
	Display_Value	MCC = 1, LOC=6
Clock	Init	MCC = 1, LOC=4
	Increment	MCC = 1, LOC=4

For the taxon *class,* the measure Size2 proposed by Li and Henry [Li 93] is calculated for each class in the software system. The results of the calculation are summarized below.

Class	Measure
Instrument	Size2 = 2+2 = 4
Speedometer	Size2 = 1+1 = 2
Gauge	Size2 = 1+1 = 2
Clock	Size2 = 3+3 = 6
Graf_Gauge	Size2 = 3+1 = 4
Chronometer	Size2 = 0+1 = 1
Accu_Clock	Size2 = 1+1 = 2

For the taxon *inheritance,* the measure is DIT (Depth of Inheritance Tree) proposed by Chidamber and Kemerer [Chidamber 94]. The results of the calculation are summarized below.

Class	Measure
Instrument	DIT = 2
Gauge	DIT = 1
Clock	DIT = 1

For the taxon *coupling and uses,* the measure is CCM (total Count of the number of accesses to other Classes, accesses by other classes and the number of cooperating classes) proposed by Chen [Chen 93]. The results of the calculation of CCM are summarized below.

Speedometer, Gauge and Clock access Instrument
    count is 3

Graf_Gauge accesses Gauge
    count is 1

Chronometer and Accu_Clock access Clock
    count is 2

    CCM = 6

For the taxon *system,* the measure is SC1 (total number of edges in the hierarchy graph for the system), proposed by Abreu and Carapuça [Abreu 94]. From Figure 4-4, the total number of edges in the hierarchy graph for the system is six. (Simply count the arrow heads.)

    SC1 = 6

Teaching Considerations

Prerequisites

Measuring object-oriented software products may be discussed in any classroom setting in which the students have a basic background in a high-level object-oriented programming language and some knowledge of elementary descriptive statistics; this level of familiarity would normally be found, for instance, in students who have had the CS1-CS2 programming courses plus an elementary statistics course. Of course, the material will be more meaningful to students who have some experience with object-oriented systems of greater size and complexity, either in industry or in a course that has a system development project.

Recommended Module Uses

In a Software Engineering Lecture Course

Objectives and Content: In a typical software engineering course, sometimes as little as one class hour is devoted to measurement topics. In this case the instructor can hope to do little more than make students aware of the importance of the problems associated with measurement and will not have time to discuss the various object-oriented methods. As an overview of measurement concepts, one good starting point is the SEI document by Gary Ford [Ford 93], which may be used by the instructor in preparing lectures; or short segments may be copied and distributed to the students for reading. For a one hour lecture, Chapter 1 of Fenton's book provides a good background for preparing a lecture on the need for measurement. If time permits, some of the more mathematical topics from Chapter 2 can be injected into the lecture [Fenton 91, pp.1-22].

However, a deeper understanding of measurement and measuring object-oriented software products, the importance of measurement in software engineering, and the various object-oriented methods would seem to merit a more extensive study than can be accomplished in a one-hour lecture. If four to five class hours are available, the following topics could reasonably be covered.

Lecture 1 - Use the material in Section 1 of this module; cover the material in Section 1.3.2 very lightly.

Lecture 2 - Use the material in Section 2 of this module, with good coverage of one object-oriented design method and light coverage of the other methods. The details of Section 2.1 can be omitted. (Remember that students have seen objects and a method before.)

Lecture 3* - Use the material in Section 3 of this module. Carefully cover the taxonomy of Section 3.1 and choose typical measures (two or three) from each taxon to discuss in some detail. Point out to students the problems with various interpretations of 'coupling.'

*This lecture can be expanded to two hours very easily by more thoroughly discussing the measures for the *taxon method*, which includes the classic measures—Halstead's software science measures, McCabe's cyclomatic complexity measure, lines of code, and function points.

Lecture 4 - Use the material in Section 4 of this module. Choose either of the two examples and discuss the measures chosen.

At the end of the unit a student should:

- Know the importance of measurement in the life cycle and the importance of planning with a goal (objective) in mind.
- Understand and be able to explain the GQM paradigm, use the notation of at least one object-oriented method, and understand the need to isolate the features of an object-oriented software product for measurement.
- Know of the existence of measures for the five main features of object-oriented software products and be able to select and combine some of these into a measures suite for a specific goal and set of questions.

If more time is available, the content could be expanded in one of two ways. First, more detail on the object-oriented methods could be discussed, such as notation or software tools that are provided by some vendors for a specific method. Second, any of the measures suggested in Section 3 can be discussed in greater detail.

Resources: As an overview of measurement concepts a good starting point is, again, the SEI document by Gary Ford [Ford 93], which may be used by the instructor in preparing lectures. Ford's material will also lead the instructor to sources on the engineering issues of measurement. A second good reference, other than Fenton's book, is the Putnam text whose first chapter discusses various views of complexity [Putnam 92]. Although Putnam's approach is aimed at process improvement, Chapter 1 of his text helps motivate the need for measurement.

An exercise that may be interesting is to ask students to review one or more tools, based on papers such as the ones in this module's bibliography or on product literature from commercial vendors. I have found that students who have experienced the drudgery of graphic design of an object-oriented system can easily appreciate the benefits of tool support;

for an inexperienced class, however, this sort of exercise may be less useful.

In a Software Metrics Lecture Course

Objectives and Content: In a typical software metrics course, object-oriented measures are not taught. The instructor usually makes students aware of the importance of measurement and spends the remainder of the course covering both process measures and the classic product measures. I have taught such a 'classical' course twice in 1990 and 1991 using Conte's text [Conte 86], supplemented with two lectures on non-parametric statistics [Sachs 84]. The primary reason today for the omission of object-oriented measures is that there are simply no texts that include this topic. For a three-credit-hour lecture course on software measures, Fenton's book provides a good foundation for measurement and the classical process and product measures [Fenton 91]. The objective is to modernize this course by injecting six lectures on object-oriented software measures into the course at week three or four. These lectures can come directly from this module, plus some supplemental readings from the bibliography. The following material on object-oriented software measures would be the best:

Lecture 1 - Use the material in Section 2 of this module with good coverage of one object-oriented design method and light coverage of the other methods. (Remember that students have seen objects and an object-oriented method before.) The details of Section 2.1 can be omitted. Duplicate the list of terminology and distribute it to the class.

Lecture 2 - Use the material in Section 3 of this module. Carefully cover the taxonomy of Section 3.1. Duplicate the taxonomy and distribute it to the class. Choose typical measures (at least three) from each taxon to discuss in some detail. Point out to students the problems with various interpretations of 'coupling.' Save the taxon *method* to discuss in Lecture 3.

Lectures 3 & 4 - This lecture covers the taxon *method* and can take two hours very easily by more thoroughly discussing the measures for the taxon *method*, which includes the classical measures—Halstead's software science measures, McCabe's cyclomatic complexity measure, lines of code, and function points. An interesting in-class exercise is to display a program written in the firm's (or school's) programming language for all to see and have the students each compute Halstead's effort measure [Conte 86, p.84]. (This involves some subjective decision-making, possibly group consensus, and generates many questions.)

Lectures 5 & 6 - Use the material in Section 4 of this module. The two examples given in Section 4 are simple, and the Ada95 example is missing some code for the base class methods. The missing code consists of combinations of assignment statements and output

statements, neither of which adds complexity to these methods. Adding your own methods to any of the base classes can create an example that may be more meaningful for your students. Choosing other measures for any of the taxa can mold these examples to fit your classroom needs.

At the end of this unit of lectures a student should:

- Know and understand the importance of measurement in the life cycle and the wide applicability of measurement as it also relates to object-oriented software products.
- Understand and be able to use the notation of at least one object-oriented method, and understand that measuring the features of an object-oriented software product encompasses more issues than merely a software module.
- Know of the existence of measures for the five main features of object-oriented software products, and be able to select and combine some of these into a measures suite for a specific goal and set of questions.

If more time is available, the content could be expanded in one of two ways. First, more detail on the object-oriented methods could be discussed, such as notation or software tools that are provided by some vendors for a specific method. Second, any of the measures suggested in Section 3 can be discussed in greater detail.

Resources: As an overview of measurement concepts, one good starting point is the SEI document by Gary Ford [Ford 93], which may be used by the instructor in preparing lectures; or, short segments may be copied and distributed to the students for reading. A second good reference, other than Fenton's book, is the Zuse text which discusses various views of complexity from a mathematical viewpoint [Zuse 90]. The Zuse text could be used with great success in a graduate-level measurement course.

Project Suggestions

The specification and development of a code analyzer tool can be a challenging and interesting project for a student team in a project course. The possibilities are extensive and can range from quite easy projects to systems of considerable difficulty.

Some possibilities are (in order of increasing difficulty):

- A program to count lines of code based on a set of guidelines developed using the Park recommendations and instructor (or student) specifications.
- A program to calculate McCabe's cyclomatic complexity measure for a program (module).
- A program to count the number of coupling relationships in a large software system.

- A program to compute the Halstead effort measure for a program (module).

Bibliography: Index by Author

Measuring Object-Oriented Software Products

Bibliography

This bibliography is organized into five sections for ease of reference. These sections in order of occurrence are:

- Articles Related to Object-Oriented Measures
- Early Seminal (Much Quoted) Works on Measures
- Textbooks and Papers on Measurement and Topics Closely Related to Measurement
- Textbooks on the Object-Oriented Approach
- Texts on Mathematics and Statistics Relating to Measures

Articles Related to Object-Oriented Measures

Abreu94

Abreu, Fernando B. & Carapuça, Rogério. "Candidate Metrics for Object-Oriented Software within a Taxonomy Framework." *Journal of Systems Software 26*, 1 (July 1994): 87-96.

The authors provide a taxonomy for metrics of object-oriented products and processes. This taxonomy, TAPROOT, deals with both product and process metrics plus some "hybrid" metrics that measure both. The author's taxonomy is based on a Cartesian product of the two vectors: (design, size, complexity, reuse, productivity, quality) and (method, class, system). This produces eighteen possible cells into which a metric can reside. TAPROOT is presented as a starting point from which further refinement and verification can follow.

Aksit92

Aksit, Mehmet & Bergmans, Lodewijk. "Obstacles in Object-Oriented Software Development," pp. 341-358. *Proceedings: OOPSLA Conference.* Vancouver, B.C., October 18-22, 1992. New York, NY: ACM Press; Reading, MA: Addison-Wesley, 1992.

Based on the results of some pilot studies, the authors have formed their own viewpoint of object-oriented methods and have documented some obstacles. The authors state that each phase in object-oriented software development can be subdivided into three sub-components: preparatory work, structural

relations, and object interactions. A short summary of state-of-the-art object-oriented methods follows the subdivision taxa.

Archer95

Archer, Clark B. & Stinson, Michael C. *Object-Oriented Software Measures:* (CMU/SEI-95-TR-002). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1995.

This technical report presents guidelines for classifying object-oriented software measures and reports the measures that are presented in the literature relative to this classification scheme. The authors propose a common terminology for the various object-oriented methods to reduce misunderstandings in reporting future measures. This report also includes an extensive annotated bibliography of current work on object-oriented measures.

Banker91

Banker, Rajiv D.; Kauffman, Robert J.; & Kumar, Rachina. "An Empirical Test of Object-based Output Measurement Metrics in a CASE Environment." *Journal of Management Information Systems 8*, 3 (Winter 1991): 127-150.

This 23-page article begins by reporting studies that indicate the use of CASE without having measurement programs in place. The authors' main thrust is the issue of output measurement in a CASE environment.

Their comments on function points (FP) are

  FP components do not follow naturally from an object-based CASE environment.

  Application of FP to CASE-generated code is subjective and inconsistent.

  Albrecht's original FP weights need to be re-calibrated for CASE tools.

  The usual Technology-Complexity-Factor (TCP) adjustment for FP may need revised for CASE since TCP is based on 3GL development.

The authors propose a short-form variation of FP called Raw-Function-Counts and two new object-based output measures, Object-Counts and Object-Points. The authors statistically validate the various metrics to estimate effort, and their results are significant. These proposed measures worked well in the CASE environment created by the ICE software. The authors conclude, "Since objects were found to match project managers' mental model of the functionality of software developed with object-based CASE, information about objects would be useful to promote improved software development process control."

Byard94

Byard, Cory. "Software beans: Class metrics and the mismeasure of software." *Journal of Object-Oriented Programming 7*, 5 (September 1994): 32-34.

This non-technical article discusses "why measure software," "class metrics," and "mismeasurement." The author comments, "class metrics do not measure complexity, do not measure the size of an application, and do not measure the quality of software." Class metrics "are indicators of programming style." The author concludes, "The key is not measurement, but process"; and, "developing new measures that analyze implementation vocabulary complexity, module cohesion and coupling, and development progress will help."

Chen93

Chen, J-Y. & Lum, J-F. "A New Metric for Object-Oriented Design." *Information of Software Technology 35*, 4 (April 1993): 232-240.

The authors use Basili's Goal-Question-Metric model to develop metrics for complexity for object-oriented design. The authors propose eight metrics that are identifiable at the design stage:

1. operation complexity metric
2. operation argument complexity metric
3. attribute complexity metric
4. operation complexity metric

5. class coupling metric
6. cohesion metric
7. class hierarchy metric
8. reuse metric

Metrics 1 through 3 are subjective in nature; metrics 4 through 7 involve counts of features; and metric 8 is a boolean (0 or 1) indicator metric. To validate these metrics, the authors conduct an experiment involving six "experts" whose subjective class scores are regressed against the eight metrics. The resulting regression equation is used to score future object classes. The paper does not report the original data, the complete SAS output, or the criteria that the "experts" use to measure complexity.

Chidamber91

Chidamber, Shyam R. & Kemerer, Chris F. "Towards a Metrics Suite For Object Oriented Design," pp. 197-211. *Proceedings: OOPSLA'91*. Phoenix, AZ, October 6-11, 1991. New York, NY: ACM SIGPLAN Notices, 1991.

The authors propose six metrics that they evaluate relative to seven of Weyuker's properties. The authors' objective is to propose metrics that are not language specific. They introduce measures that capture some features such as coupling, cohesion, complexity, scope, and methods (defined as responses to possible messages).

Chidamber94

Chidamber, Shyam & Kemerer, Chris F. "A Metrics Suite for Object-Oriented Design." *IEEE Transactions on Software Engineering 20*, 6 (June 1994): 476-493.

The authors use the theoretical base for ontological principles proposed by Bunge as a means of establishing a basis upon which to discuss the object-oriented metrics suite. Much of the material in the first four pages is the same as in their earlier paper in 1991. The authors define six metrics and evaluate them with respect to six of Weyuker's nine properties. They propose six metrics for object classes:

1. Weighted Methods per Class (WMC).
2. Depth of Inheritance Tree (DIT).
3. Number Of Children (NOC), number of immediate subclasses subordinate to a class in the hierarchy.
4. Coupling Between Object classes (CBO).
5. Response For a Class (RFC), cardinality of the set of all methods that can be invoked by some method in the class.
6. Lack of Cohesion Of Methods (LCOM), the number of method pairs whose similarity is zero minus the counts of the method pairs whose similarity is not zero.

These metrics are based on three assumptions: the inheritance tree is full, two classes can have a finite number of identical methods, and certain counts of features are random variables that are identically and independently distributed.

Churcher95

Churcher, Neville & Sheppard, Martin J. "Towards a Conceptual Framework for Object-Oriented Software Metrics." *Software Engineering Notes 20*, 4 (April 1995): 69-75.

The authors caution that software measures for object-oriented systems present significantly greater challenges than their conventional counterparts. They propose a set of terms to serve as a basis for comparison of models of object-oriented systems. They emphasize the problems arising from different interpretations of coupling and uses. They summarize by stating "it seems premature to proceed with the speculative development of specific metrics due to the absence of a satisfactory framework for their validation."

Coppick92

Coppick, Chris J. & Cheatham, Thomas J. "Software Metrics for Object-Oriented Systems," pp. 317-322. *Proceedings: ACM CSC '92 Conference.* Kansas City, MO, March 3-5, 1992. New York, NY: ACM Press, 1992.

The authors extend the Halstead metric and McCabe metric to object-oriented software design. The authors' examples are in LISP Flavors. An undefined tool (code not included) is applied to LISP source code, and the usual software science metrics are computed. The authors count the number of methods and observe that increased abstraction reduces programming effort. Nothing concrete is done with McCabe's metric.

Gowda94

Gowda, Raghava G. & Winslow, Leon E. "An Approach for Deriving Object-Oriented Metrics," pp. 897-904. *Proceedings: IEEE 1994 National Aerospace and Electronics Conference.* Dayton, OH, May 23-27, 1994. Los Alamitos, CA: IEEE Computer Society Press, 1994.

The authors comment, "The object-oriented metrics proposed so far seem to concentrate on the design of a single class or the class structure and ignore the overall design of the system and program." They propose a classification scheme for object-oriented metrics with the five categories of system metrics, subsystem metrics, class metrics, object metrics, and reusability metrics. They discuss and contrast each of the methodologies of Rumbaugh and Wirfs-Brock. The authors claim to have a list of metrics that can be applied to some of the phases of each methodology. Although the authors actually list some features of the phase and methodology that can be measured, they do not indicate how to measure the feature.

Laranjeira90

Laranjeira, Luiz. "Software Size Estimation of Object-Oriented Systems." *IEEE Transactions on Software Engineering 16*, 5 (May 1990): 510-522.

The author presents a size estimation model that takes advantage of the characteristics of object-oriented systems and their specification. He also provides a confidence interval for the expected system size. COCOMO is applied in this setting to produce cost estimates.

Lee92

Lee, Yen-Sung; Liang, Bin-Shiang; & Wang, Feng-Jian. "Some Complexity Metrics for Object-Oriented Programs Based on Information Flow," pp. 302-310. *Proceedings: CompEuro.* Paris-Ivry, France, May 24-27, 1993. Los Alamitos, CA: IEEE Computer Society Press, 1993.

The authors use Weyuker's nine properties as a basis of evaluation. They define four metrics: method complexity (MC), class complexity (CC), hierarchy complexity (HC), and program complexity (PC). These measures are based on various forms of the basic model:

size*(input coupling + output coupling)^2

None of the proposed metrics satisfy Weyuker's seventh property.

Li93

Li, Wei & Henry, Salley. "Maintenance Metrics for the Object Oriented Paradigm," pp. 52-60. *Proceedings: First International Software Metrics Symposium.* Baltimore, MD, May 21-22, 1993. Los Alamitos, CA: IEEE Computer Society Press, 1993.

The authors state that metrics for the object-oriented paradigm have yet to be studied. Since terminology varies among object-oriented programming languages, the authors consider the basic components of the paradigm as objects, classes, attributes, inheritance, method, and message passing. They propose that each object-oriented basic concept implies a programming behavior. They include six metrics from Chidamber [Chidamber 91]:

Depth of Inheritance Tree (DIT) Coupling Between Objects (CBO)

Number Of Children (NOC) Response For Class (RFC)

Lack of Cohesion Of Class (LCOM) Weighted Method per Class(WMC)

The authors construct a Classic-Ada metric analyzer to collect metrics from Classic-Ada design and source code. They define five additional metrics to complete the modeling:

Data Abstraction Coupling (DAC) Number of Methods (NOM)

# of semicolons per class (Size1) # of methods + # attributes (Size2)

Message Passing Coupling (MPC)

A regression analysis is used with Change = number of lines changed in the artifact's history (classes) as the dependent variable. The authors' analysis of the results reveals that the metrics chosen (all 10) can predict the number of changes. There is no individual breakdown of which of these metrics is significant in the prediction.

Lieberherr89

Lieberherr, Karl J. & Holland, Ian M. "Assuring Good Style for Object-Oriented Programs." *IEEE Software 6*, 5 (September 1989): 38-48.

The authors put forward a simple law, the Law of Demeter, that encodes the ideas of encapsulation and modularity in an easy-to-follow form for object-oriented programmers. The law has two forms: class and object forms. The class form comes in two versions: minimization and strict versions.

Class minimization version - *Minimize the number of acquaintance classes over all methods.*

Class strict version - *All methods may have only preferred-supplier classes.*

Objects - *All methods may have only preferred-supplier objects.*

The motivation behind the Law of Demeter is to ensure that the software is as modular as possible. The law effectively reduces the occurrences of nested message sending and simplifies the methods.

Moreau90a

Moreau, Dennis R. & Dominick, Wayne D. "A Programming Environment Evaluation Methodology for Object-Oriented Systems: Part I - The Methodology." *Journal of Object-Oriented Programming 3*, 1 (May/June 1990): 38-52.

The authors set forth three objectives for their research (paraphrased below):

1. Establish an evaluation methodology to measure impact of object-oriented design on the software development process.

2. Establish domain-specific problem decomposition and solution guidelines to support comparisons of object-oriented approaches.

3. Perform verification of object-oriented metrics.

The principles of the proposed method are based on user activities, are environment-independent, and are based on well-constructed experiments. The authors claim that the method is extensible, captures the structural object-oriented aspects of the software, and provides for the automatic capturing of the metrics-related data. The authors include Halstead's little n and big N metrics and McCabe's cyclomatic complexity metrics, along with two measures that are based on object-oriented features, a graph of the source and destination of all messages, and an inheritance lattice. This paper provides a clear overview of a method for measuring object-oriented software.

Moreau90b

Moreau, Dennis R. & Dominick, Wayne D. "A Programming Environment Evaluation Methodology for Object-Oriented Systems: Part II - Test Case Application." *Journal of Object-Oriented Programming 3*, 3 (September/October 1990): 23-32.

In this companion article to their article above, Moreau and Dominick discuss a refinement of the objectives set forth previously into theoretical, methodological, developmental, and evaluative components. The methodology is applied in an interactive graphics application domain. The test case was completed in 11 phases:

1- Identify applications domain {interactive graphics editor}

2- Identify test development systems {C & C++}

3- Identify development paradigms {GKS for C & object-oriented for C++}

4- Identify metrics {those in Moreau [Moreau 1990a]}

5- Identify and classify development activities {three separate tasks}

6- Establish evaluative criteria {Basili's direct cost/quality criteria}

7- Develop environment independent experiments

8- Prepare environments {no functional differences}

9- Develop environment-specific experiments {8 subjects, 4 in each experimental group}

10- Perform experiments

11- Analyze results {non-parametric Wilcoxon statistics P=0.07}

The authors state, "This research has formally established the primary metric data definitions that completely characterize the unique aspects of object-oriented software systems, including the inheritance lattice and message graph."

Page-Jones92

Page-Jones, Meilir. "Comparing Techniques by Means of Encapsulation and Connascence." *Communications of the ACM 35*, 9 (September 1992): 147-151.

The author contrasts structured design and object-oriented design, and proposes that object-oriented designs can be measured by the new property, connascence. Connascence is a generalization of coupling and cohesion, which the author defines as "Two software elements A and B are connascent iff there is at least one change that could be made to A that would necessitate a change to B in order to preserve overall correctness." Page-Jones claims this concept is applicable to object-oriented design and advises, "Eliminate any unnecessary connascence and then minimize connascence across encapsulation boundaries by maximizing connascence within encapsulation boundaries."

Poulin94

Poulin, Jeffrey S. & Brown, David D. "Measurement-Driven Quality Improvement in the MVS/ESA Operating System," pp. 17-25. *Proceedings: Software Metrics Symposium*. London, U.K., October 24-26, 1994. Los Alamitos, CA: IEEE Computer Society Press, 1994.

This paper describes experiences, quality initiatives, models, and metrics used to obtain quantifiable results in a large, complex software system. Although no object-oriented metrics were actually developed or computed, this paper shows that the introduction of object-oriented design and the construction of high quality reusable frameworks played a critical role in defect reduction.

Sharble93

Sharble, Robert C. & Cohen, Samuel S. "The Object-Oriented Brewery: A Comparison of Two Object-Oriented Development Methods." *SIGSOFT Software Engineering Notes 18*, 4 (April 1993): 60-73.

This paper reports on research to compare the effectiveness of two methods for the development of object-oriented software. The two methods compared are responsibility-driven methods and data-driven methods. Each of the methods was used to develop a model of the same system. The authors use a suite of object-oriented metrics to collect measures of each model. The model developed with the responsibility-driven method was found to be less complex, to have less coupling between objects, and to have more cohesion

within objects. The research produced three new metrics that can be useful for measuring object-oriented designs.

WAC - Weighted Attributes per Class.

NOT - Number of Tramps (number of extraneous parameters in signatures of methods of a class.

VOD - Violations of the Law of Demeter.

Taenzer89

Taenzer, David; Ganti, Murthy; & Podar, Sunil. "Object-Oriented Reuse: The Yo-yo Problem." *Journal of Object Oriented Programming.* (September/October 1989): 30-35.

The authors review two basic approaches to software reuse, construction, and inheritance, and present some basic problems and conflicts between encapsulation and inheritance. They discuss the basic styles for reuse of construction and subclassing. Based on their own experiences in reuse, the authors give examples of message control trees. This discussion leads to the definition of the "Yo-yo" problem, where resolutions of a message goes up and down the message tree.

Tegarden91

Tegarden, David P.; Sheetz, Steven D.; & Monarchi, D.E. "Effectiveness of Traditional Metrics for Object-Oriented Systems," pp. 359-368. *Proceedings 25th Hawaii International Conference on System Sciences 4.* Kauai, HI, January 7-10, 1992. Los Alamitos, CA: IEEE Computer Society Press, 1991.

The authors begin by quoting Moreau: "traditional metrics are inappropriate for object-oriented systems for several reasons..." [Moreau 90]. This paper addresses two questions, 'Can existing metrics developed for structured systems be used as effective measures of object-oriented systems?' and 'Can certain unique aspects of object-oriented systems be measured by traditional metrics?' They discuss the traditional SLOC, Halstead metrics, and the cyclomatic metric and these metric's potential use in the object-oriented setting. The authors conclude, "The use of the traditional metrics may be appropriate for the measurement of the complexity of object-oriented systems. Even though the order of magnitude of the traditional metrics may be suspect, the directionality seems to be correct."

Williams93

Williams, John D. "Metrics for Object Oriented Projects," pp. 13-18. *Proceedings: Object Expo Euro Conference.* London, U.K., July 12-16, 1993. New York, NY: ACM SIGS Publications, 1993.

The author poses the question, "Why metrics?" The answer, he says, is in both project management metrics and software development metrics. He proposes a "3db" curve for monitoring project progress. Neither the 3, the d, nor the b is defined. For software development, the author suggests using counts of "uses," counts of the number of base classes (classes that represent reused code), counts of stand-alone classes, and counts of the number of

"contains" relationships in a class. He comments, "depending on how deep a class is in the inheritance tree, it may have many 'hidden' members and methods."

Yap93a

Yap, L.M. & Henderson-Sellers, Brian. "Consistency Considerations of Object-Oriented Class Libraries." (Research Report 93-3). Sydney, Australia: University of New South Wales, 1993.

Yap93b

Yap, L.M. & Henderson-Sellers, Brian. "A Semantic Model for Inheritance in Object-Oriented Systems." pp. 28-35. *Proceedings: Senenth Australian Software Engineering Conference.* Sydney, Australia, September 27-October 1, 1993. Edgecliff, NSW: IREE Australia Publications, 1993.

The authors examine the various forms of inheritance in object-oriented software engineering. With the goal of organizational consistency, they propose a semantic model in which object classes are divided between domain classes and implementation classes.

Early Seminal (Much Quoted) Works on Measures

Albrecht79

Albrecht, A J. "Measuring application development productivity," pp. 83-92. P*roceedings: IBM Applications Development Joint SHARE/GUIDE Symposium.* Monterey, CA, October 14-17, October 1979. Chicago, IL: IBM, 1979.

This is the seminal work on function points. Albrecht's intent is to measure the amount of functionality in a software product based on either the coded product or a structured specification document. As stated by the author "The general approach is to count the number of external user inputs, inquiries, outputs, and master files delivered by the development project. These factors are the outward manifestations of any application. They cover all the functions in an application." The weights that Albrecht originally assigned to the four external attributes was four, five, four, and ten, respectively.

Halstead77

Halstead, M. H. *Elements of Software Science.* North-Holland, NY: Elsevier Publishing, 1977.

This is the original early work on measuring coded software products based on lexical issues of the product, such as numbers of operators, operands, unique operators, and unique operands. The theory for both the length metric and the volume metric is based on the lexical features of the program. Halstead's length measure is the total occurrences of operators and operands;

while the volume measure is the product of the length measure and the vocabulary measure (the sum of the number of unique operators and operands). Halstead's effort measure is based on the principles of cognitive psychology and a subjectively determined constant called the Stroud Number.

Henry81

Henry, Sallie & Kafura, Dennis. "Software Structure Metrics Based on Information Flow" *IEEE Transactions on Software Engineering 7*, 5 (September 1981): 510-518.

The authors propose that controlling system structure improves external quality, and propose a means of measuring information flow between system components. They propose measures for procedure complexity, module complexity, and module coupling. Their complexity measures are based on fan-in and fan-out.

McCabe76

McCabe, T.J. "A Complexity Measure." *IEEE Transactions on Software Engineering 2*, 4 (April 1976): 308-320.

McCabe's cyclomatic complexity metric is the first of the attempts at measuring complexity. The metric is based on the features of a directed graph representation of the software product.

Textbooks and Papers on Measurement and Topics Related to Measurement

Basili88

Basili, Victor & Rombach, H. Dieter. "The TAME Project: Towards Improvement-oriented Software Environments." *IEEE Transactions on Software Engineering 14*, 6 (June 1988): 758-773.

The authors introduce a set of software engineering and measurement principles based on twelve years of analyzing both software products and software engineering processes. The Goal-Question-Metric paradigm is proposed as a mechanism for formalizing the characterization, planning, construction, analysis, learning, and feedback tasks for software projects. They use this paradigm in the TAME (Tailoring A Measurement Environment) project at the University of Maryland. This is a solid paper that could be used as required reading in both measurement and software engineering classes.

Card90

Card, David L. & Glass, Robert L. *Measuring Software Design Quality.* Englewood Cliffs, NJ: Prentice-Hall, 1990. ISBN 0-135-68593-1.

This short paperback text (104 pages plus appendices and references) is quite readable. The book proposes a small set of measures (referred to as "primitive design metrics") that are centered around design quality. The authors' intent is to provide the practitioner with criteria for improving software designs to promote productivity, quality, and maintainability. Most of the examples and data come from a structured design environment with FORTRAN as the language.

Carleton92

Carleton, Anita; Park, Robert E.; Goethert, Wolfhart; Florac, William,; Bailey, E.; & Pfleeger, Sally. *Software Measurement for DOD Systems: Recommendations for Initial Core Measures.* (CMU/SEI-92-TR-19, ADA258305). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1992.

This report presents the recommendations for a basic set of software measures that DoD organizations can use to help and manage the acquisition, development, and support of software systems. The concept is the use of checklists to create and record measurement descriptions and reporting specifications. These checklists provide a mechanism for obtaining consistent measures from project to project and for communicating unambiguous measurement results.

Conte86

Conte, S.D.; Dunsmore, H.E.; & Shen, V.Y. *Software Engineering Metrics and Models.* Menlo Park, CA: Benjamin/Cummings, 1986. ISBN 0-805-32162-4

This text presents the classical product measures, classical models of process, and the product and process measures currently available in the late 1980s. The authors include a chapter on experimental design and basic statistical inference. They present a set of model evaluation criteria that practitioners should find useful. They examine effort from two viewpoints, macro and micro environments, and include the classical studies that are associated with each of these environments. This text would serve well in a traditional senior-level software measures class.

Fenton91

Fenton, Norman E. *Software Metrics, A Rigorous Approach.* London: Chapman & Hall, 1991. ISBN 0-412-40440-0.

This text is solid and well written. Chapter 1 motivates the discipline. Chapters 2 through 6 provide a coherent framework for the many diverse activities that comprise software metrics. Among these are measurement theory, design of experiments, and data collection. Chapters 8 through 13 cover process measures, product measures, and resource measures. The author has provided an extensive, partially annotated bibliography.

Ford93

Ford, Gary. *Lecture Notes on Engineering Measurement for Software Engineers.* (CMU/SEI-93-EM-9, ADA266959). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1993.

This material's goal is to facilitate teaching software engineering measurement. Materials are provided to support three lectures: introduction to engineering measurement, measurement theory, and software engineering measures. These materials include lecture notes suitable for class handouts and informational material for instructors. I recommend this as required reading for measurement instructors.

Humphrey90

Humphrey, Watts. *Managing the Software Process.* Reading, MA: Addison-Wesley, 1990. ISBN 0-201-18095-2.

This book is a byproduct of a project to provide guidance to the DoD for selecting software contractors. The end result is the development of the Software Engineering Institute's Capability Maturity Model (CMM[SM]).[1] This text discusses this model and provides guidance for the need to measure both process and product.

Jones86

Jones, Capers. *Programming Productivity.* New York, NY: McGraw-Hill, 1986.

The author summarizes the first 30 years of industrial and commercial programming. The first two chapters of this four-chapter book are about the science of measurement and serve as an introduction to the topic of measurement. In the third chapter, the author isolates 20 factors, supported by historical data, that have affected programming productivity.

1. The language used	11. Maintenance
2. Program size	12. Reuse (modules & design)
3. Personnel experience	13. Code generators
4. Requirements	14. 4GLs
5. Complexity of program & data	15. Separation of dev. locales
6. Use of structured methods	16. Defect detection & removal
7. Program class	17. Documentation
8. Program (application area)	18. Prototyping
9. Tools & environment	19. Project teams & organization
10. Enhancing existing systems	20. Morale & compensation of staff

Chapter 4 explores the intangible factors, which are not readily quantifiable, that affect productivity. These factors include size of staff and enterprise, stability during the project, training for staff and users, computing facilities,

[1] CMM is a service mark of Carnegie Mellon University.

legal issues, project measurement mechanisms, outsourcing, project dynamics, and user participation among all of these. This is a good book for the beginning software engineer. Jones has a second edition of this work in publication.

Mills88

Mills, Everald E. *Software Metrics*. (SEI Curriculum Module SEI-CM-12-1.1, ADA236140). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1988.

This curriculum module introduces the most commonly used software metrics and models as of 1988. Both process measures and models are covered along with product measures. Mills closes the module with some recommendations for the implementation of a metrics program. and current trends in software metrics.

Park92

Park, Robert E. *Software Size Measurement: A Framework for Counting Source Statements* (CMU/SEI-92-TR-20, ADA258304). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1992.

This technical report presents guidelines for defining, recording, and reporting two frequently used measures of software size: lines of code and logical source statements. Park proposes a general framework for constructing size definitions and uses it to derive operational methods for reducing misunderstandings in measurement results.

Putnam92

Putnam, Lawrence H. & Myers, Ware. *Measures for Excellence.* Englewood Cliffs, NJ: Yourdon Press, 1992. ISBN 0-13-567694-0.

This text is aimed at practitioners. The text is divided into two parts, the first part dealing with software behavior and the second part, procedural in nature, applying the patterns of behavior to real projects. The text includes a wealth of accumulated real-world experience and relates some statistics based on the data accumulated by Putnam's company, Quantitative Software Management (QSM). This could serve as a text for a one-semester measurement course.

Symons88

Symons, Charles. "Function Point Analysis: Difficulties and Improvements." *IEEE Transactions on Software Engineering 14*, 1 (January 1988): 2-11.

The author critically reviews Albrecht's function point analysis, proposes ways of overcoming the weaknesses identified, and validates by experimentation the proposed improvements. Some criticisms are that FPs underweight systems that are complex internally and FPs are not

"summable."   The author proposes the "Mark II" formula for information processing component size in unadjusted function points which is:

UFP = NI*WI + NE*WE + NO*WO

where

NI = number of input data elements

WI = weight of an input data type

NE = number of entity-type references

WE = weight of an entity-type reference

NO = number of output data element types

WO = weight of output data element type

Symons determines a set of weights from 12 systems and recalibrates these weights to match Albrecht's original UFP values for systems under 500 FPs. He concludes that

· Mark II involves an understanding of entity analysis, no conventions   yet.

· Mark II has fewer variables to count, but more technical factors to consider.

· Albrecht's FP is not a technology-independent measure of system size and neither is Mark II, since a change in technology involves recalibrating.

· FP analysis works for business applications, but may not work well for scientific or technical applications.

Waguespack87

Waguespack, Leslie J, Jr. & Badlani, Sunil. "Software Complexity Assessment: An Introduction and Annotated Bibliography." *Software Engineering Notes 12*, 4 (October 1987): 52-71.

The authors provide an introduction to software complexity and provide an exhaustive list of nineteen categories of complexity research. The works listed in the article cover the years 1974-1987, plus one entry from 1967. Some five hundred works are listed in the form [Lastname##] where ## is the last two digits of the year, and a hundred of these were selected for the annotated bibliography. The annotated bibliography contains the complete reference citation and the original abstract (or an excerpt from the work which portrays the author's intent) followed by the annotation.

Wang85

Wang, A.S. & Dunsmore, H.E. "Early Software Size Estimation: A Critical Analysis of the Software Science Length Equation and a Data-Structure-Oriented Size Estimation Approach," pp. 211-222. *Proceedings: Third Symposium on Empirical Foundations of Information and Software Science.* Rosklide, Denmark, October 21-24, 1985. New York, NY: Plenum Publishing Co., 1985.

The authors address early size estimation by emphasizing the weaknesses of the current size estimation metrics in 1985. They conjecture that program

size can be estimated as a function of some other measurable quantities related to the program. Empirically, data from Pascal programs suggest that the Halstead length equation is not suitable for predicting the size of large Pascal programs. The authors found that the count of the VAR (the number of unique variables) is an acceptable size estimation. Experimental results yield:

$S = 102 + 5.31*VAR$ as an estimate with $r = 0.94$ and mean MRE = 0.30

Based on these results, early estimation of program size can be improved at the end of the design stage by using the VAR count. The authors caution that these are "lab" results, and software that was produced in the lab was not nearly as large as that produced in industry.

Weyuker88

Weyuker, Elaine. "Evaluating Software Complexity Measures." *IEEE Transactions on Software Engineering 14*, 9 (September 1988): 1357-1365.

Weyuker establishes a standard for software measures in this seminal article. She states the conditions for a measure as follows:

"All the measures we consider depend only on the syntactic features of the program."

$P + Q$ means that programs P and Q halt on the same input.

P;Q means that P is augmented by Q (a concatenation).

The measure of P is denoted by $| P |$.

The nine properties of measures:

1. $(\exists P) (\exists Q) \ (\, | P | \ \Box \ | Q | \,)$.

2. Let c be a number ≥ 0. Then there are finitely many programs of complexity c.

3. There are distinct programs P and Q such that $| P | = | Q |$.

4. $(\exists P) (\exists Q) \ (P + Q \ \text{and} \ | P | \ \Box \ | Q | \,)$.

5. $(\forall P) (\forall Q) \ (\, | P | \ \leq \ | P ; Q | \ \text{and} \ | Q | \ \leq \ | P ; Q | \,)$.

6. $(\exists P) (\exists Q) (\exists R) \ (\, | P | \ = | Q | \,) \& (\, | P ; R | \ \Box \ | Q ; R | \,)$ and $(\exists P) (\exists Q) (\exists R) \ (\, | P | \ = | Q | \,) \& (\, | R ; P | \ \Box \ | R ; Q | \,)$.

7. There are program bodies P and Q such that Q is formed by permuting the order of the statements of P; and $| P | \ \neq \ | Q |$.

8. If P is a renaming of Q, then $| P | \ = \ | Q |$.

9. $(\exists P) (\exists Q) \ (\, | P | \ + \ | Q | \ < \ | P ; Q | \,)$.

Zuse90

Zuse, Horst. *Software Complexity Measures and Methods.* Berlin: Walter de Gruyter & Co, 1990. ISBN 3-110-12226-X.

This is the most comprehensive coverage of software complexity measures available in 1990. The text covers the issue of "metric versus measure,"

discusses measurement, and discusses the various ways that data can be classified. The author includes at least ninety measures that have appeared in the literature (mostly European sources). The text is strongly recommended as a reference for researchers and instructors.

Textbooks on the Object-Oriented Approach

Booch94

Booch, Grady. *Object-Oriented Analysis and Design, Second Edition.* Redwood City, CA: Benjamin/Cummings, 1994. ISBN 0-805-35340-2.

This is a complete text for learning the essence of the object-oriented approach. It covers the notation of the Booch method, discusses analysis and design strategies, and contains an extensive bibliography. Grady summarizes the text very well in the preface, "We first present a graphic notation for object-oriented analysis and design, followed by its process. We also examine the pragmatics of object-oriented development—in particular, its place in the software development life cycle and its implications for project management." The text is a good reference book and a good text for an upper-level undergraduate class.

Coad91

Coad, Peter & Yourdon, Edward. *Object-Oriented Analysis, Second Edition.* Englewood Cliffs, NJ: Yourdon Press, 1991. ISBN 0-136-29981-4.

The authors cover object-oriented analysis in a straight-forward manner and introduce an object-oriented analysis (OOA) methodology consisting of five steps: identifying classes and objects, identifying structures, identifying subjects, defining attributes, and defining services. All of these items are combined into an "object diagram," which resembles a dataflow diagram combined with an entity-relationship diagram. The book's strength is the discussion of management issues that emerge from using object-oriented techniques.

Coleman94

Coleman, Derek; Arnold, Patrick; Bodoff, Stephanie; Dollin, Chris; Gilchrist, Helena; Hayes, Fiona; & Jeremaes, Paul. *Object-Oriented Development The Fusion Method.* Englewood Cliffs, NJ: Prentice-Hall, 1994. ISBN 0-133-38823-9.

Bertrand Meyer summarizes this text: "This book could have been entitled *Putting it all together*...The great merit of the method described here is that it starts at the beginning of the software construction process and accompanies the reader all the way to the end." The text targets software engineers and project managers with some knowledge of the object-oriented approach. The authors propose the Fusion method, which integrates existing approaches to provide a direct route from the requirements specification to the

implementation. The book also contains reference material on the Fusion method.

Firesmith93

Firesmith, Donald G. Object-Oriented Requirements Analysis and Logical Design. New York, NY: John Wiley & Sons, 1993. ISBN 0-471-57806-1.

The text's goal is to "provide the profession of software engineering with the necessary concepts, models, notations, methods, and knowledge needed to effectively develop large, complex software applications using a practical, state-of-the-art, object-oriented method." The author presents the ADM3 method to achieve this goal. This book could be used in advanced undergraduate or graduate-level classes. Extensive references and a five-section appendix (one section covers MIL-STD-2167A) are two of the book's features.

Henderson-Sellers92

Henderson-Sellers, Brian. *A Book of Object-Oriented Knowledge.* Englewood Cliffs, NJ: Prentice-Hall, 1992. ISBN 0-130-59445-8.

The author summarize this book in his preface, "What is this book about? I have tailored it to be a basic introduction to the object-oriented approach to software engineering, emphasizing analysis and design at the expense of the syntax of object-oriented programming languages." This paperback book contains 169 full-page exhibits that can be blown up for presentations. Also, the book contains a short annotated bibliography of books (18 of these) on the object-oriented approach that were written in the years 1986-1991.

Henderson-Sellers94

Henderson-Sellers, Brian & Edwards, J.M. *Book Two of Object-Oriented Knowledge: The Working Object.* Riverwood, NSW Australia: Prentice-Hall Ligare Pty Ltd, 1994. ISBN 0-130-93980-3.

This text is the sequel to Henderson-Sellers' 1992 book, and focuses on analysis and design presenting the MOSES methodology as a means of providing a "seamless transition" across the development life cycle. The authors clearly state that the book is not a cookbook for MOSES, but the methodology is described fully. In Chapter 7, Section 13, the authors include a quality evaluation activity that incorporates software metrics, and Chapter 10 is devoted to object-oriented "metrics." This text can be used for a one-semester course on object-oriented technology.

Jacobson92

Jacobson, Ivar; Christerson, Magnus; Jonsoon, Patrik; & Overgaard, Gunnar. *Object-Oriented Software Engineering A Use Case Driven Approach.* Reading, MA: Addison-Wesley, 1992. ISBN 0-201-54435-0.

This text serves as a good introduction to the object-oriented technique. It presents object-oriented software engineering (OOSE) as a new methodology that emphasizes the interaction of the user with the system and emphasizes the problem domain. The authors goal for the text is to present a coherent picture of how to use object-orientation in system development so as to make

it accessible to both practitioners in the field and students with no previous knowledge of system development. The text contains clear examples of the object-oriented approach at all levels of software development and certainly achieves the author's goal.

Rumbaugh91

Rumbaugh, J. et al. *Object-Oriented Modeling and Design.* Englewood Cliffs, NJ: Prentice-Hall, 1991. ISBN 0-136-29841-9.

This text is a popular, but older, coverage of the subject. The authors propose a complete methodology, the object modeling technique (OMT), which covers analysis, design, and implementation. The authors contrast their OMT with structured analysis and design and with Jackson's structured development method. For those of us who are familiar with the procedure-oriented techniques, the text provides a smooth transition to object-oriented techniques. This material has been updated but the update has not been published as of May, 1995.

Wirfs-Brock91

Wirfs-Brock, Rebecca; Wilkerson, B; & Wiener, L. *Designing Object-Oriented Software.* Englewood Cliffs, NJ: Prentice-Hall, 1991. ISBN 0-136-29825-7.

The authors define the object-oriented approach and provide a complete coverage of object-oriented principles. They emphasize a responsibility-driven viewpoint of analysis and design that emphasizes clients and servers. They also suggest that quality of design can be measured by counts of the number of classes, the number of subsystems, the number of contracts per class, and the number of abstract classes. The diagrams are clear and reinforce the material. This was one of the first books to focus on design.

Yourdon94

Yourdon, Ed. *Object-Oriented System Design, An Integrated Approach.* Englewood Cliffs, NJ: Prentice-Hall, 1994. ISBN 0-136-36325-3.

This text is portrayed by the author as "a synopsis and an integration of several popular object-oriented development methods, with particular emphasis on object-oriented analysis and design." The book is broken down into six parts. Part 1 (Introduction) motivates object-orientation. Part 2 (Management Issues) covers reuse and management of object-oriented projects. Parts 3 & 4 (Object-Orientation Analysis and Design) covers just that. Part 5 (CASE for Object-Orientation) discusses CASE tools and their vendors. Part 6 (How To Get Started) covers how to introduce the object-oriented approach into the organization. This is a "practitioner-oriented" text.

Texts on Mathematics and Statistics Relating to Measures

Mansfield63

Mansfield, Maynard. *Introduction to Topology*. Princeton, NJ: Van Nostrand, 1963.

This is a classic text on topology. This small book (116 pages) covers the basics of point set topology at the undergraduate level, and is a source of discussion for metrics and metric spaces.

Sachs84

Sachs, Lothar. *Applied Statistics: A Handbook of Techniques, Second Edition*. New York, NY: Springer-Verlag, 1984. ISBN 0-387-16835-4.

This text is an excellent reference for statistical techniques and the concept of measuring phenomena so that they can be evaluated statistically. The text contains a wide range of tables of value to statisticians. It is also a good source of non-parametric statistical procedures.

Siegel88

Siegel, Sidney & Castellan, N. John. *Nonparametric Statistics for the Behavioral Sciences, Second Edition*. New York, NY: McGraw-Hill, 1988. ISBN 0-07-057357-3.

This is a welcome edition to Siegel's earlier text of the same title, which was written in 1956. The authors include an extensive discussion of measurement scales in Chapter 3. Castellan has included 5 BASIC programs in Appendix II to calculate some of the more difficult statistics. The text contains a wealth of well-constructed examples to assist the reader in understanding non-parametric statistical inference.